UNIVERSITETET I OSLO
Institutt for informatikk

Unntakshåndtering i modellering og programmering

Masteroppgave

Oddleif Halvorsen

23. mai 2005

Sammendrag

Oppgaven ”Unntakshåndtering i modellering og programmering” tar for seg utfordringen med å lage mekanismer for unntakshåndtering i UML 2 sine sekvensdiagrammer. Unntakshåndtering i sekvensdiagrammer er så langt ikke del av UML standarden, men svært sentralt for å gjøre modeller mer oversiktelige. Problemstillingen gikk ut på hva unntakshåndtering er og om det er noen forskjell på unntakshåndtering i programmering og modellering. Den mest sentrale delen av problemstillingen var så å utforme mekanismer for unntakshåndtering til UML sine sekvensdiagrammer og som skulle resultere i en presis semantikk. I tillegg skulle jeg utforme en metodikk for bruk av mekanismene.

Oppgaven ble gjennomført ved bruk av litteratur søk og gjennom et case studie av et Java basert pratesystem. Case studiet ble brukt for å utforme forslag til mekanismer til unntakshåndtering, samt utprøving av mekanismene og metodikken.

Resultatet av gjennomføringen er et forslag til mekanismer hvor sekvensdiagrammer går over flere logiske plan og bruker UML 2 Testing Profile sin default mekanisme for å ta imot unntak. Dette forslaget medførte en mer kompleks kontrollflyt for sekvensdiagrammer, men gav gode resultater med tanke på å skape et klart visuelt skille mellom en normalflyt og unntakshåndtering.

Forord

Dette er min Masteroppgave i Informatikk, utført ved Institutt for Informatikk ved Universitet i Oslo. Det er en kort masteroppgave og har blitt gjennomført vårsemesteret 2005. Faglig veileder har vært Førsteamanuensis Øystein Haugen, Institutt for Informatikk ved Universitetet i Oslo.

Jeg vil gjerne rette en spesiell takke til følgende personer som har bidratt med støtte og inspirasjon gjennom arbeidet med Masteroppgaven.

Først min veileder Øystein Haugen, for mange interessante diskusjoner og gode råd rundt oppgaven.

Line Merethe Rubach, for hennes hjelp med rapporten sin form.

Mine foreldre Liv og Leif, og mine søstrer Rita og Mona, for deres oppmuntringer og støttende ord under arbeidet.

Oddleif Halvorsen
Blindern, mai, 2005

Innhold

Figurliste

Innledning

UML kom i oktober 2004 i sin siste utgave, versjon 2.0. Versjon 2 inneholder blant annet en stor endring av sekvensdiagrammene. Man kan nå uttrykke både løkker, alternativer, referanser, parallellitet og så videre. Noe som ikke kan utrykkes er unntakshåndtering (eng.: *exception handling*).

Unntakshåndtering har lenge blitt brukt i programmeringsspråk. Der har det blitt brukt for å skille kode for unntakshåndtering fra kode for normalflyten. Videre har unntakshåndteringen tilbudt en løs kobling mellom det å flagge (eng.: *raise*) et unntak (eng.: *exception*) og det å motta et unntak. Dette visuelle skillet gjør koden mer leselighet, og gir dermed også en bedre oversikt over både normalflyten og unntakene.

Jeg vil i denne oppgaven se på hva unntakshåndtering er, om det er forskjell på hvordan unntakshåndtering bør brukes i modellering kontra hvordan det brukes i programmering. Deretter vil jeg presentere språkmekanismer for unntakshåndtering i sekvensdiagrammer og en presis semantikk for disse. Til slutt vil jeg presentere en metodikk for bruk av mekanismene for unntakshåndtering.

Da sekvensdiagrammer gjerne brukes for å utrykke asynkron meldingsutveksling vil også semantikken til unntakshåndtering i sekvensdiagrammer ta utgangspunkt i asynkron meldingsutveksling. Dette valget gjør at jeg velger å ikke ta direkte utgangspunkt i unntakshåndteringen til språk som C++, Java og C#, da disse er laget primært for synkron unntakshåndtering.

1. Unntakshåndtering

Unntakshåndtering (eng.: *exception handling*) er ikke noe nytt fenomen som kom med C++. Forslag til mekanismer for unntakshåndtering som del av programmeringsspråk ble presentert så tidlig som i 1975 av John. B. Goodenough. Jeg vil her presentere hva flere sentrale personer sier om unntakshåndtering.

Selv om jeg i innledningen sa at unntakshåndteringen i språk som blant annet C++ ikke er interessant i forhold til sekvensdiagrammer, så mente jeg ikke teorien. Teori rundt unntakshåndtering på et høyt nok nivå er noe som omfatter både modellering og programmering.

1.1. John B. Goodenough

Jeg begynner med å se på hva John B. Goodenough i 1975 presenterte i artikkelen: ”*Exception Handling: Issues and a Proposed Notation*”.

1.1.1. Hva er unntak?

For å kunne snakke om unntakshåndtering bør man ha en klar formening om hva unntak (eng.: *exceptions*) er. Unntak er ikke en bestemt ting, unntak er en bestemt oppførsel. Man kan altså ikke nødvendigvis gå inn i et dataprogram og se etter unntak, men man må se etter oppførsel som kan lede til unntak. Dette gjør unntak blir uangripelige.

Dette gjenspeiles også i Goodenough (1975) sin definisjon av hva unntak er:

> *“Of the conditions detected while attempting to perform some operation, exception conditions are those brought to the attention of the operation’s invoker.”* (Goodenough, 1975:1)

Goodenough (1975) presenterer her flere sentrale begreper når det gjelder unntak og unntakshåndtering. For det første så spesifiserer han at unntak må kunne evalueres opp mot en gitt betingelse (eng.: *condition*) som blir oppdaget under utføringen av en operasjon.

Goodenough (1975) sier videre at unntak skal håndteres av klienten (*the operation’s invoker*). Det vil si at hvis en funksjon A kaller en funksjon B, og B flagger (eng.: *raise*) et unntak, så skal A håndtere dette unntaket. Dette betyr at den som oppdager unntaket ikke kan vite hvordan det skal håndteres.

Et neste poeng Goodenough (1975) gjør, når det gjelder håndtering av unntak er: ”*Of the conditions detected* ***while attempting*** *to perform some operation, …”*. Om man legger vekt på “while attempting” ser en at budskapet er at hvis operasjonen ikke får til det den prøver på, så informerer man klienten. Uten å si det direkte kommer det her fram at unntak skal brukes til å beskrive hendeleser som går ut over normalflyten.

Et siste poeng er at definisjonen ikke bare omfatter feil, noe som ordet gjerne assosieres med, men definisjonen legger også til rette for at unntakshåndtering kan brukes til overvåking av kjøringer. Man kan altså se for seg at en funksjon informerer en klient om ulike tilstander underveis i en kjøring ved hjelp av unntak.

Det er også nettopp dette som er en vesentlig del av Goodenough(1975) sin definisjon av unntakshåndtering. Han definerer følgende områder for unntakshåndtering:

1. Å tillate håndtering av operasjoner som feiler.
2. Beskrive omstendighetene rundt et resultat.
3. Å overvåke en operasjon.

Når det gjelder unntak som tillater håndtering av operasjoner som feiler deler Goodenough (1975) disse opp i 2 deler:

- Range failure
- Domain failure

Range failure er typisk feil på output assertions. Det har altså skjedd noe feil under kjøring, for eksempel at man prøver å dele på null. Ved å prøve å dele på null får man en *range failure*. Den andre typen, *domain failure*, går på input assertions. Altså at man har sendt inn feil type input. Et eksempel kan være at en funksjon venter på et heltall og får en tekst streng. Dette representerer en *domain failure*.

Jeg kommer ikke nå til å gå inn på de 2 andre typene av unntak, da man i senere tid har gått bort fra å håndtere disse som unntak, men nevnte de for å vise områdene Goodenough mente var relevante for unntakshåndtering.

1.1.2. Hva skjer etter at unntaket er håndtert?

Jeg har sagt at unntakshåndtering går på å håndtere oppførsel som bryter med en gitt spesifikasjon. Selve håndteringen av unntakene er det opp til den som lager systemet å definere. Etter at et unntak er håndtert dukker det opp noen nye problemstillinger, for hva skjer nå? Fortsetter man funksjonen som flagget unntaket der den slapp, eller fortsetter vi et helt annet sted?

Når det gjelder *range failures* har Goodenough(1975) spesifiserer at klienten skal ha følgende muligheter:

- Terminere/abortere operasjonen som flagget (eng.: *raised*) unntaket
- Mulighet for å fortsette operasjonen som flagget unntaket, eller prøve flere ganger hvis det er logisk.

Det med å ha muligheter for å fortsette en operasjon etter at ett unntak er flagget, er noe språk som C++ og Java ikke har mulighet til, men eksistere blant annet i språket PL/I (Ghezzi et al., 1997).[1] Muligheten for å fortsette en operasjon der den slapp var helt

[1] For en presentasjon av ulike programmeringsspråk sin unntakshåndtering se Miller et al. (1997)

sentral for Goodenough (1975) sitt ønske om å kunne bruke unntakshåndtering til å overvåke utføringen av operasjoner.

Dette med å kunne terminere eller fortsette en operasjon medfører at man kan få meget kompliserte kontrollflyter. I noen tilfeller ønsker man heller ikke å tillate at funksjonen kan fortsette etter at et unntak er kastet. Dette gjorde at Goodenough (1975) klassifiserte unntak i bestemte grupper som hver la føringer på hvordan kontrollflyten etter håndtering kunne være (om den kunne fortsettes, måtte avbrytes, osv.).

Goodenough gikk, som det vil bli beskrevet i de neste kapitlene, altså mye lengre enn hva som er gjeldene standard i dagens programmerings- og modelleringsspråk.

1.2. Stroustrup og C++

Jeg vil her presentere ett noe nyere forslag til unntakshåndtering, og et forslag som har fått stor gjennomslagskraft i moderne programmeringsspråk som C++, Java og C#.

1.2.1. Hva Stroustrup mener med unntak

De nevnte områdene til Goodenough(1975) er en ganske mye videre definisjon enn hva som er blitt den gjeldene standarden for hva unntakshåndtering er, nemlig det Bjarne Stroustrup (1997) definerte. Han definerte i sin tid C++ sin unntakshåndtering, som igjen har lagt grunnlaget for måten unntakshåndtering fungerer i nyere språk som Java og C# (Deitel et al., 2002). Stroustrup valgte å kun fokusere på punkt 1 fra Goodenough (1975), om å bruke unntakshåndtering til å håndtere feil. Stroustrup (1997) definerer unntak som følgende:

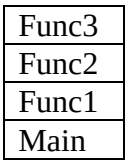
> *"Exceptions provide a way for code that detects a problem from which it cannot recover to pass the problem on to some part of the system that might be able to recover."*

Definisjonen er ikke så veldig ulike den som Goodenough (1975) bruker, men den skiller seg ut på et viktig område, Stroustrup (1997) sier eksplisitt at unntakshåndtering er til for å håndtere feil/problemer. Han går så langt som si at det er uhåndterlige feil. Stroustrup (1997) går altså bort fra Goodenough (1975) sin definisjon om at det er betingelser som skal håndteres, og ikke bare uhåndterlige feil/problemer.

En annen viktig forskjell er at han ikke legger føringer på hvem som skal håndtere unntaket. Han sier at en annen del av systemet kan prøve å håndtere det, mens Goodenough (1975) eksplisitt sier at det er klienten som skal gjøre det.

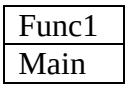
Som nevnt så holder også Java på Stroustrup sin definisjon. Java spesifikasjonen (Gosling et al., 2000) definerer ikke eksplisitt hva unntak er, men det er tydelig at også Java baserer seg på Stroustrup (1997) sin definisjon. Java praktiserer at unntak er uhåndterlige feil/problemer hvor kjøringen termineres og kontrollen gies til klienten. Metoden miste altså alle muligheter til å fortsette kjøringen og den må begynne på nytt.

Grunnen til at man mister muligheten til å fortsette kjøringer er at man håndterer flagging og fanging av unntak ved hjelp av stack-unwinding. S*tack unwinding* innebærer at unntaket søker seg tilbake gjennom kall stakken (eng.: *stack*) for å finne en passende unntakshåndterer (Stroustrup, 1997). I det en funksjon blir tatt av stakken termineres den, og man mister dermed mulighet til å fortsette der hvor funksjonen slapp. Eksempel på en kall stakk er vist i figur 1-1.



Figur 1-1 Eksempel på kall stakk

Hvis func3 kaster ett unntak så vil det resultere i at man popper stakken helt til man finner passende catch. I dette tilfellet kan vi si at func1 har en passende catch til unntaket func3 kaster. Da blir resultatet at vi popper bort func3 og func2. Dette gjør at vi blir sittende med en stakk som ser ut som vist i figur 1-2.



Figur 1-2 Kall stakken etter stack unwinding

Dette resulterer i at vi ikke har noen mulighet til å gå tilbake til dit hvor func3 kastet unntaket, fordi dette nå er fjernet fra stakken.

Videre så har Stroustrup valgt å ta bort muligheten for å fortsette en operasjon etter at eventuelle unntak er håndtert. Dette legger naturlig nok store begrensninger på mulighetene for unntakshåndtering. Ghezzi et al. (1997) argumenterer for at grunnen til at man har gått bort fra resumption til fordel for termination, er at resumption innbød til å programmere feil, eller var mye enklere å gjøre feil med. Et problem de trekker fram var at man til tider kunne ta litt lett på det å håndtere unntak ved å bare generere en tilfeldig lovlig tilstand og så fortsette kjøringen.

Jeg tror ikke det at man tok lett på saken var grunnen til at det ble valgt bort. Det er mer sannsynlig at kontrollflyten ble så komplisert at man i programmeringsspråk fikk problemer med å følge den. Derfor ble det også lett å gjøre alvorlige feil.

1.3. Oppsummering

I dette innledende kapittelet har jeg tatt for meg første delen av problemstillingen, nemlig å se på hva unntak er. Jeg har sett på hva ulike personer sier om unntakshåndtering, og ut fra dette prøvd å få fram hva unntak og unntakshåndtering er. Jeg så også på hvordan definisjonen av unntak har forandret seg over tid. Fra Goodenough (1975) vide definisjon til Stroustrup (1997) sin mer nøkterne. Hva UML og UML litteratur sier om unntak har

jeg ikke nevnt her, men kommer tilbake til dette i kapittel 3. I kapittel 3 har jeg samlet alt rundt UML og unntakshåndtering.

Kort oppsummert er unntak er en oppførsel som bryter en gitt spesifikasjon, og gjerne er noe som har en viss sammenheng med frekvens. Unntak er noe som skjer relativt sjeldent, i forhold til hva man har spesifisert som normalt.

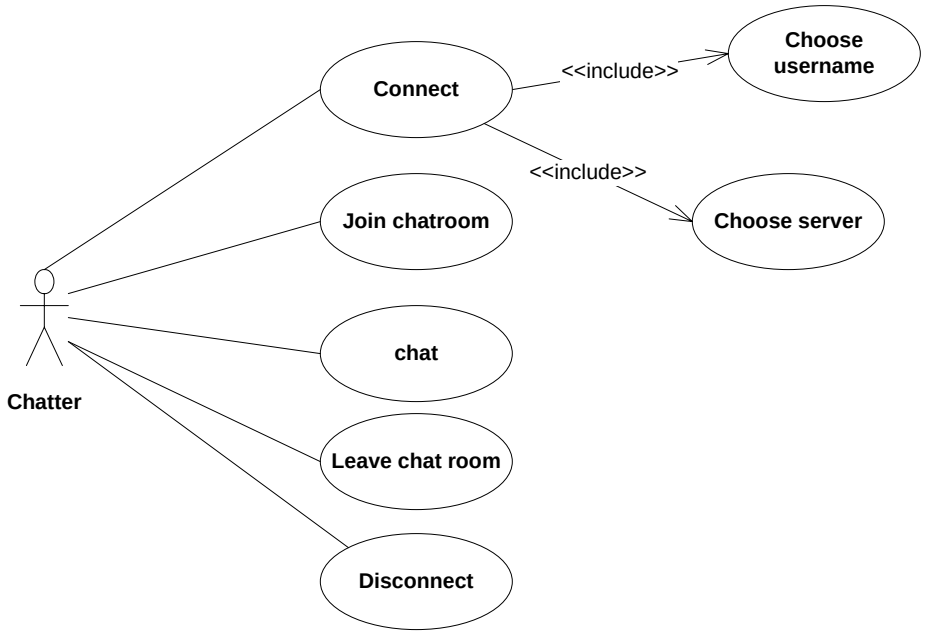
2. Case: JavaChat

Jeg vil i størsteparten av eksemplene videre bygge på spesifikasjonen til et system som beskriver et Java basert prateprogram. Jeg vil her presentere systemet sin normalflyt. Med normalflyt så mener jeg hva jeg ønsker at systemet vanligvis skal gjøre. Dette innebærer at systemet sånn som det er spesifisert i dette kapittelet ikke vil være spesielt robust, eller kanskje ikke fungere helt heller. Eksempel kan være at man prøver å slutte seg til praterom som ikke eksisterer, eller koble seg til med en annen person sitt brukernavn, osv..

For at systemet skal fungere tilfredsstillende så må det innføres unntakshåndtering, og jeg vil senere vise ulike måter å innføre dette på.

2.1. Use Case

Use Case modellen i figur 2-1 viser i grove trekk hva systemet skal tilby.



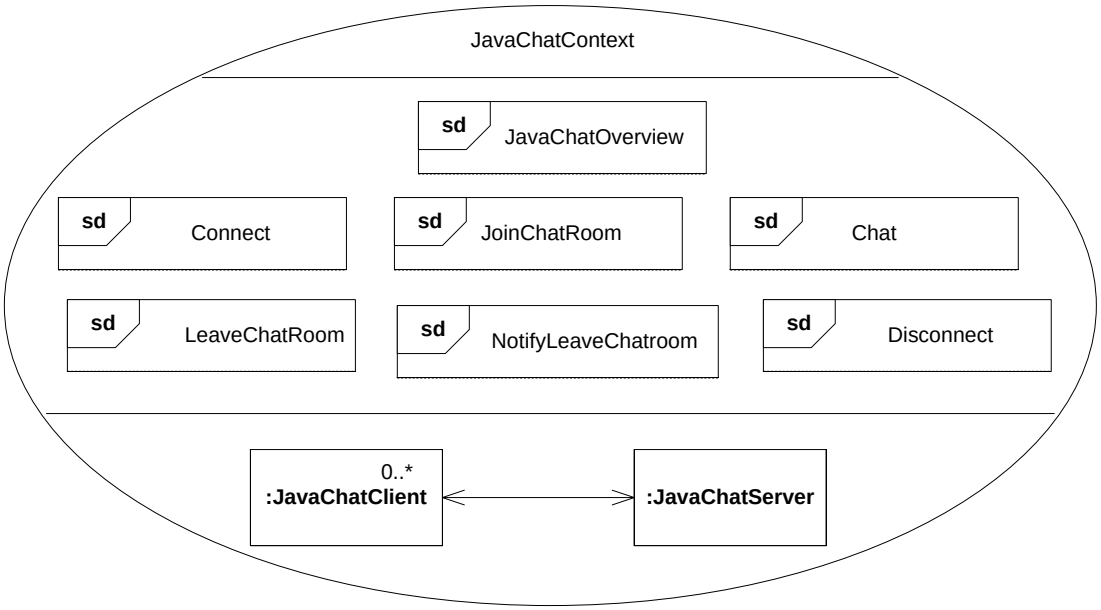
Figur 2-1 Use Case modell

Det er altså snakk om et prateprogram (eng.: *chat*). Kommunikasjon mellom brukerne (eng.: *chatters*) vil foregå i praterom (eng.: *chat rooms*). Brukerne kan koble seg til en gitt server, slutte seg til praterom, prate i praterom, forlate praterom og koble seg fra serveren.

2.2. JavaChat kontekst

I figur 2-2 vises konteksten for JavaChat systemet. Det er JavaChatServeren som blir modellert i denne gjennomgangen. Klienten, JavaChatClient, sine detaljer er her valgt

bort for å begrense omfanget. Man vil i stor grad kunne se hvordan klienten(e) vil oppføre seg ut fra hvilke meldinger som utveksles med serveren sammen med sin egen oppfatning om hvordan et prateprogram fungerer.

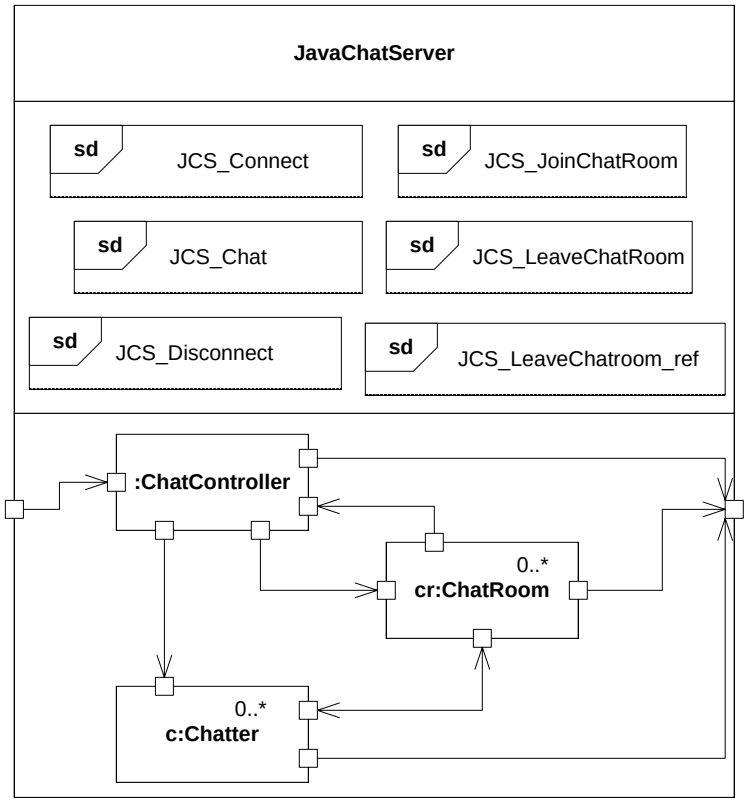


Figur 2-2 Konteksten til JavaChat

2.3. JavaChatServer strukturen

Strukturen til systemet består av en kontroller pluss en samling av praterom (*ChatRooms*) og pratere (*Chatters*).

ChatController har hovedsakelig ansvaret for å holde orden på hvilke praterom og brukere som er i systemet og å dirigere innkommende meldinger til rett mottaker. Praterommene har som oppgave å holde oversikt over hvem som er i hvert rom, og å distribuere pratemeldinger til disse. Hver Chatter har som oppgave å holde en oversikt over hvilke rom en selv er i. På denne måten er det mer effektivt å rydde opp når en bruker forlater serveren. Hvis vi ikke hadde denne Chatter strukturer ville vi for hver bruker som kobler seg fra pratesystemet være nødt til å søke oss gjennom alle praterommene for å sikre oss at brukeren var fjernet fra disse.



Figur 2-3 Strukturen til JavaChatServer

2.4. Klassediagram

Figur 2-4 viser klassene til systemet og deres assosiasjoner.

Figur 2-4 Klassediagram for JavaChatServer

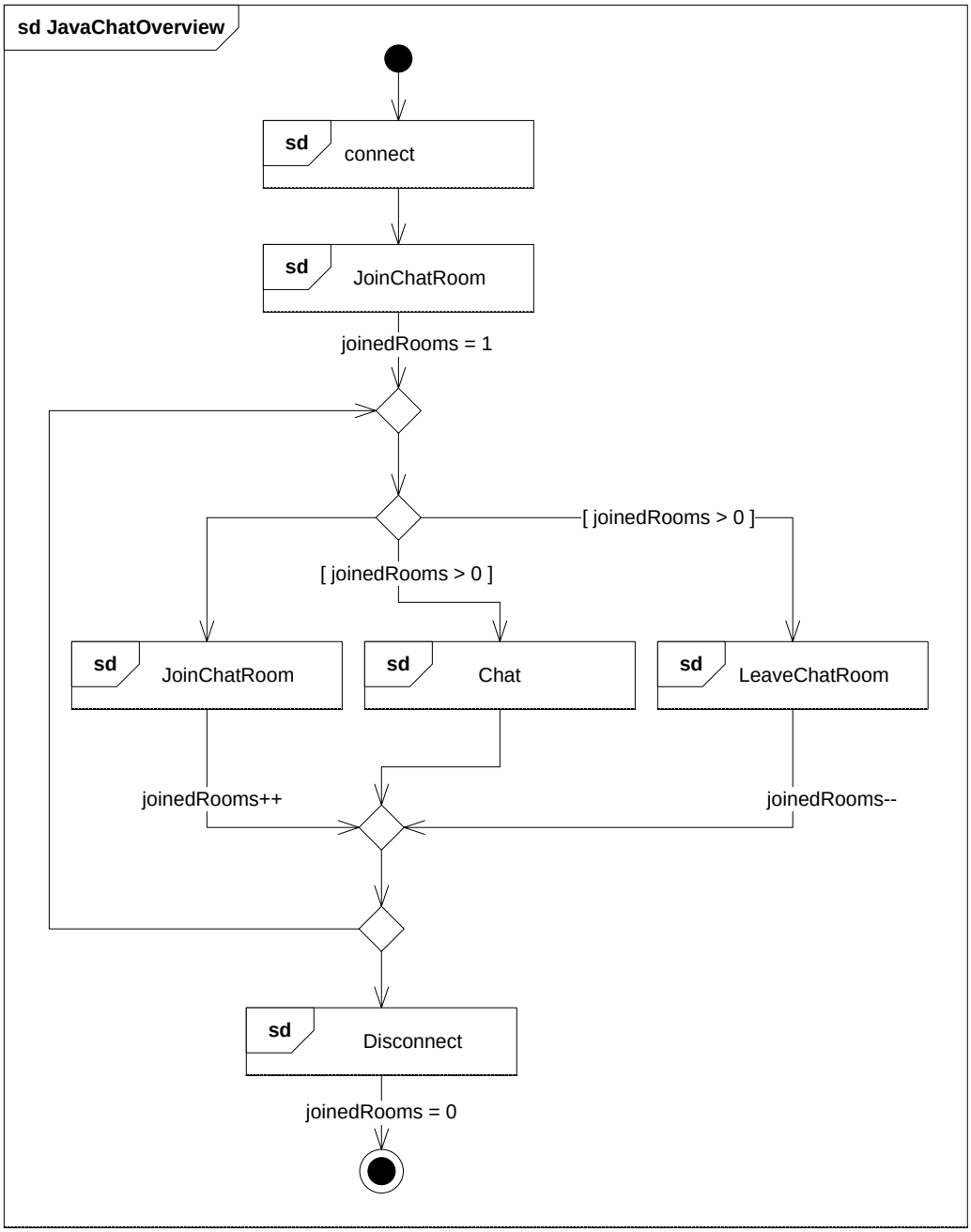
Nå som tjenestene til systemet, strukturen og ansvarsfordelingen er kartlagt kan vi gå over til å se på sekvensdiagrammene for normalflyten.

2.5. Sekvensdiagrammer

Jeg vil her beskrive sekvensdiagrammene for JavaChatServer. Disse vil, som nevnt, kun benytte seg av normalflyt for å beskrive systemet. Jeg vil senere presentere ulike måter å håndtere unntak på i noen utvalgte diagrammer.

2.5.1. JavaChatOverview

Figur 2-5 viser JavaChatOverview diagrammet. Dette diagrammet viser hvilke regler som gjelder for når man kan anvende funksjonaliteten som ligger i de ulike sekvensdiagrammene.



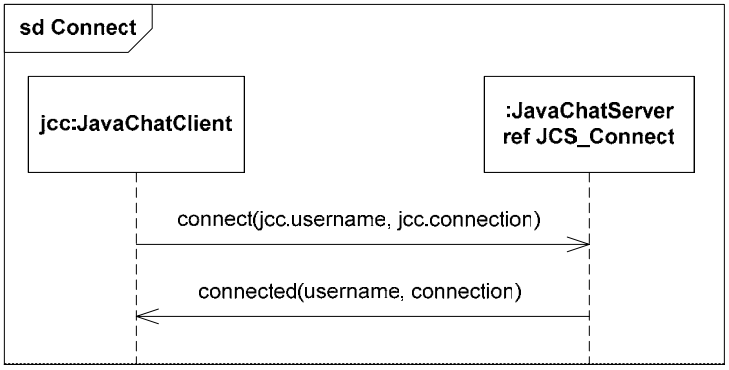
Figur 2-5 sd JavaChatOverview

Det modellen, i korte trekk sier, er at man ikke kan slutte seg til et rom før man har koblet seg tilserveren. Man kan heller ikke forlate flere rom enn man har sluttet seg til eller prate uten å være tilsluttet et rom. I tillegg så har man muligheten til å koble seg fra på et hvilket som helst tidspunkt etter tilkobling.

Noe modellen dessverre ikke viser er det at man ikke kan forlate, eller prate i rom man ikke har sluttet seg til. Dette er problematisk å vise i denne type modeller og derfor heller ikke forsøkt modellert.

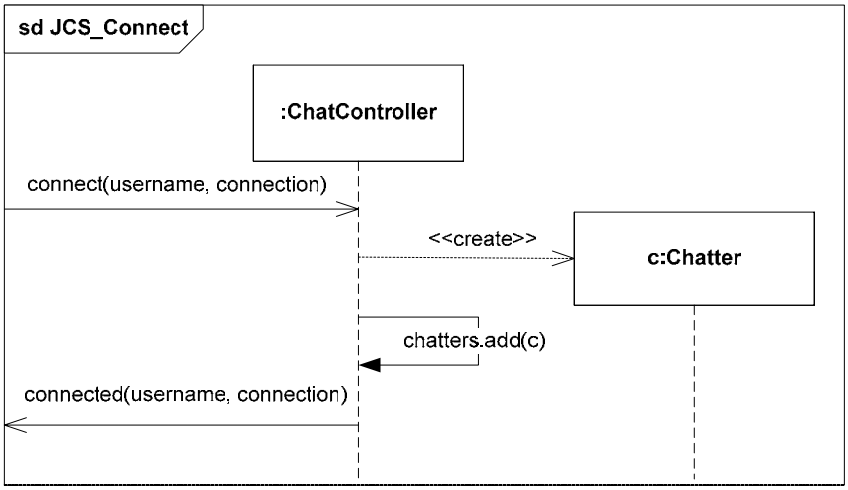
2.5.2. Connect og JCS_Connect

Sekvensdiagrammene i figur 2-6 og figur 2-7 viser hvordan en klient kobler seg opp mot serveren. Dette gjøres ved at klienten sender med ønsket brukernavn og hvordan serveren kan sende melding tilbake til klienten.



Figur 2-6 sd Connect

JCS_Connect beskriver hvordan en klient blir opprettet.

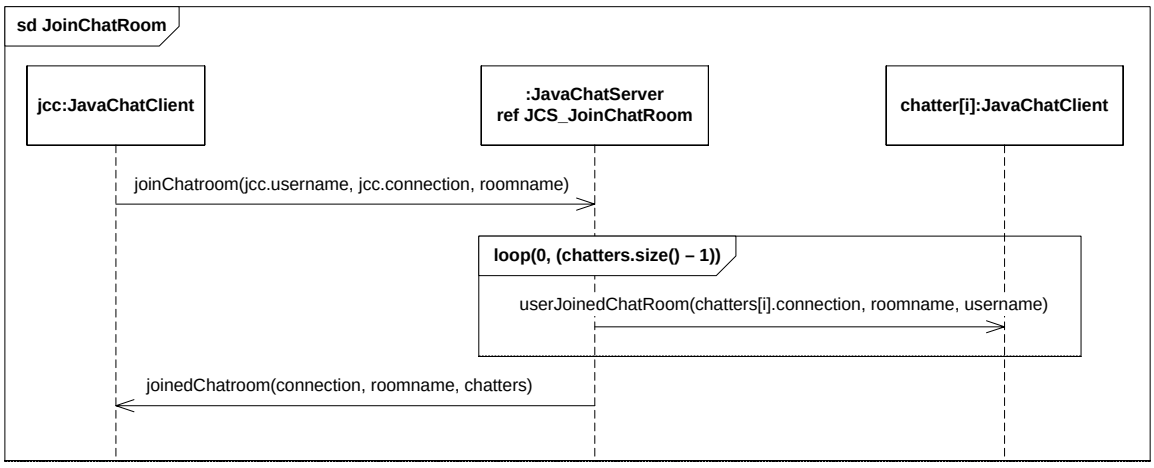


Figur 2-7 sd JCS_Connect

2.5.3. JoinChatRoom og JCS_JoinChatRoom

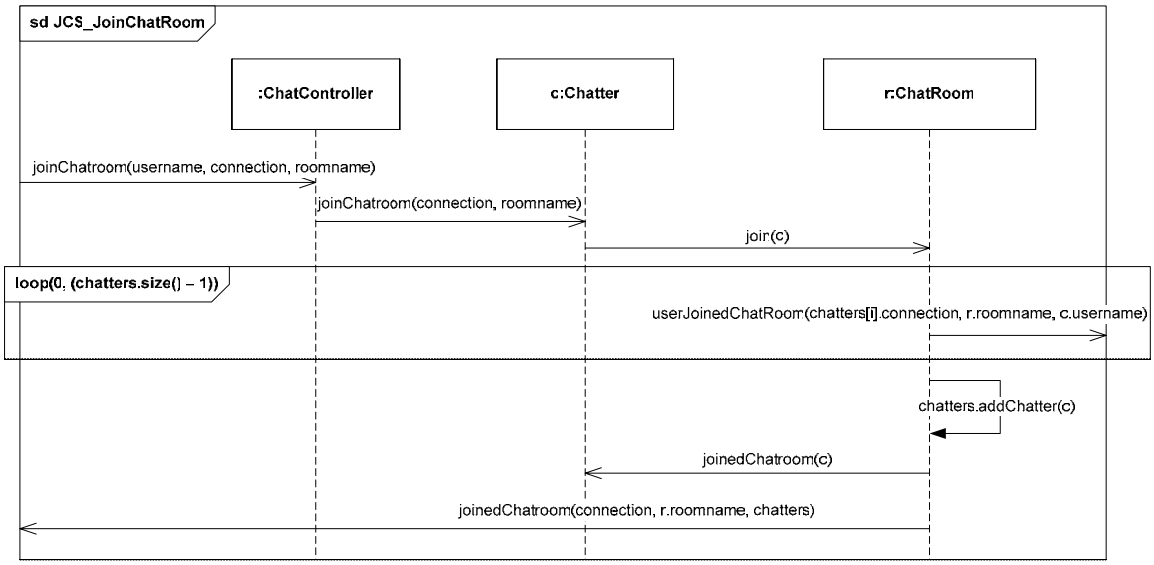
I figur 2-8 og figur 2-9 viser jeg hvordan en klient slutter seg til et praterom. I korte trekk så skjer følgende:

- Brukeren sender melding om at den vil slutte seg til ett gitt rom.
- Serveren legger rommet til listen over aktive praterom for den aktuelle brukeren
- Serveren informerer de aktive brukerne om at en ny bruker har sluttet seg til praterommet.
- Serveren legger brukeren til listen over aktive brukere i rommet.
- Serveren informerer brukeren om at den nå har sluttet seg til praterommet.



Figur 2-8 sd JoinChatRoom

I figur 2-8 så er parametere chatters i meldingen joinedRoom(...) en liste av strenger med navn på brukere som er aktive i det gitte praterommet.

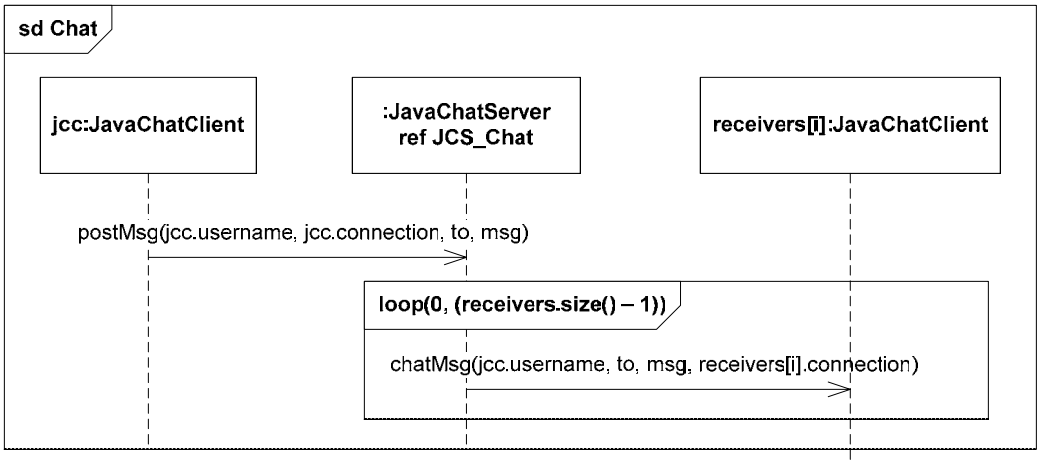


Figur 2-9 sd JCS_JoinChatRoom

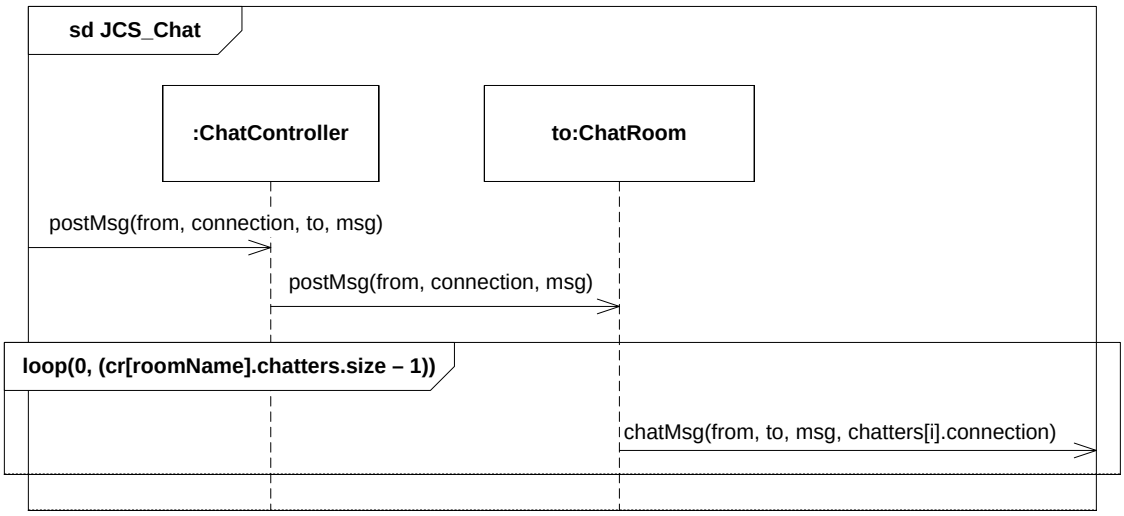
I JCS_JoinChatRoom, figur 2-9, bør en legge merke til at jeg informerer alle de aktive brukerne om den nye brukeren før jeg slutter brukeren til praterommet. Deretter sender jeg med en komplett liste over alle de aktive brukerne i praterommet til den nye brukeren.

2.5.4. Chat og JCS_Chat

Det å prate innebærer i dette systemet at en klient sender en melding til et gitt praterom, hvor så serveren distribuerer meldingen videre til alle som er aktive i praterommet.



Figur 2-10 sd Chat

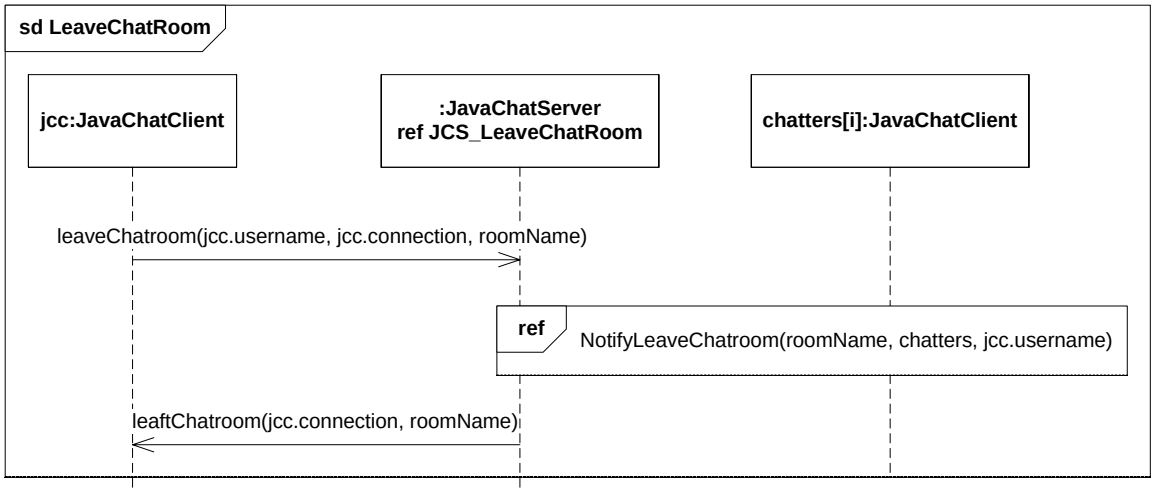


Figur 2-11 sd JCS_Chat

2.5.5. LeaveChatRoom og JCS_LeaveChatRoom

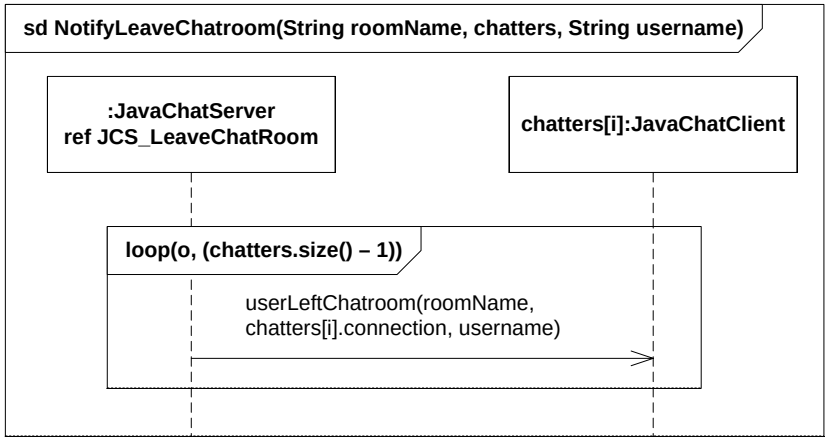
Her vises de stegene serveren utfører når en bruker forlater et praterom. Følgende skjer:

- Brukeren sier at den vil forlate et gitt praterom.
- Brukeren fjernes fra listen over aktive brukere i det gitte rommet
- Det sies så ifra til de resterende aktive brukerne i rommet om hvem som har forlatt det.
- Rommet slettes fra brukeren sin liste over praterom den er aktiv i.

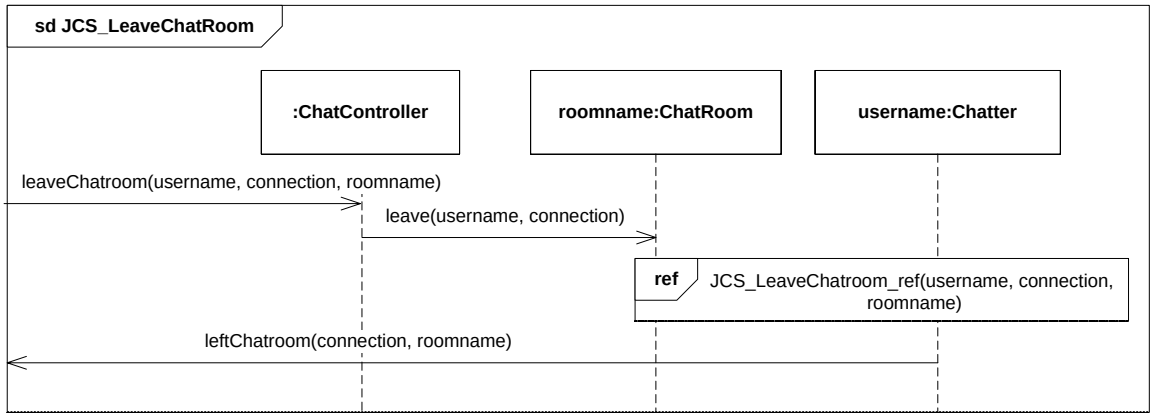


Figur 2-12 sd LeaveChatRoom

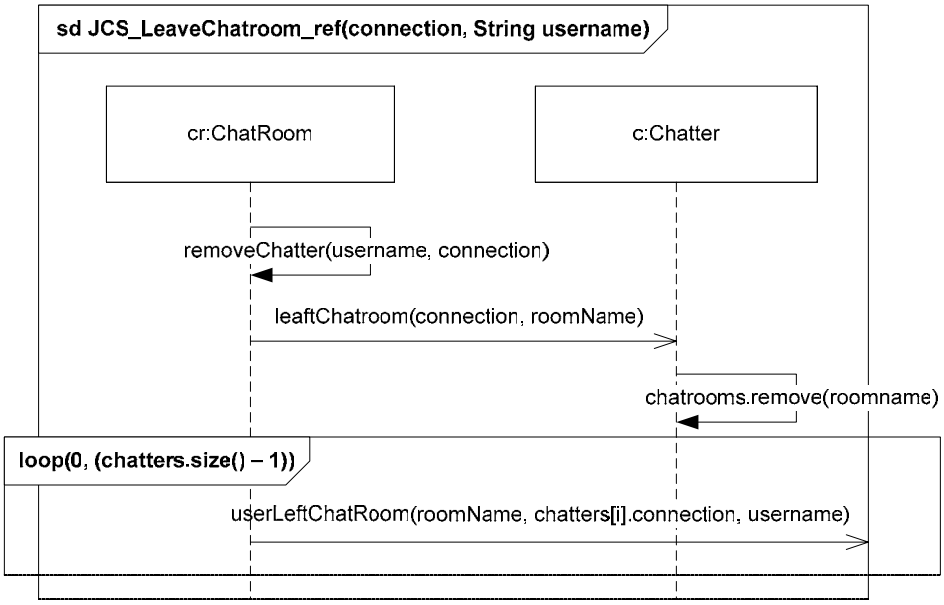
Har lagt til et diagram NotifyLeaveChatRoom som brukes av både LeaveChatRoom og Disconnect. Dette fordi det samme går igjen i diagrammene for å koble seg fra serveren.



Figur 2-13 sd NotifyLeaveChatRoom



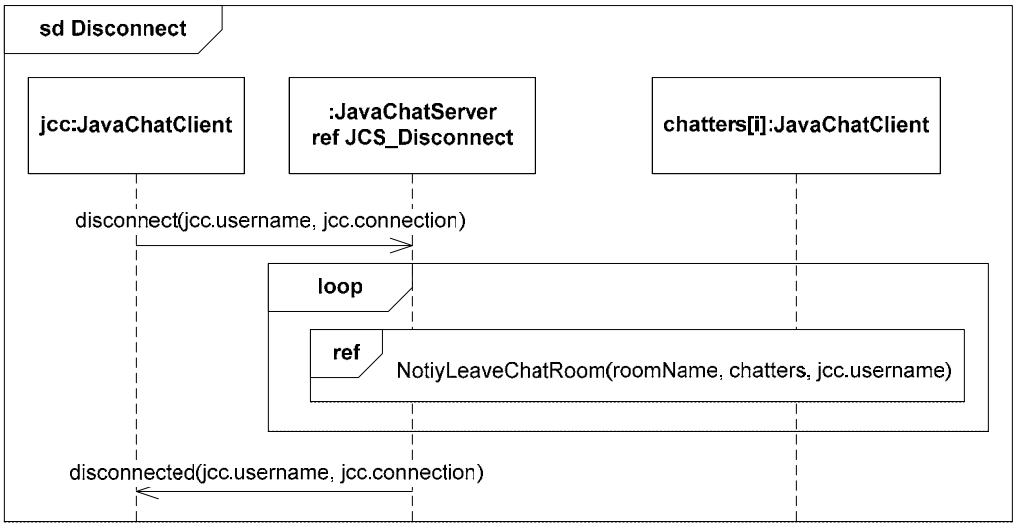
Figur 2-14 sd JCS_LeaveChatRoom



Figur 2-15 JCS_LeaveChatRoom_ref

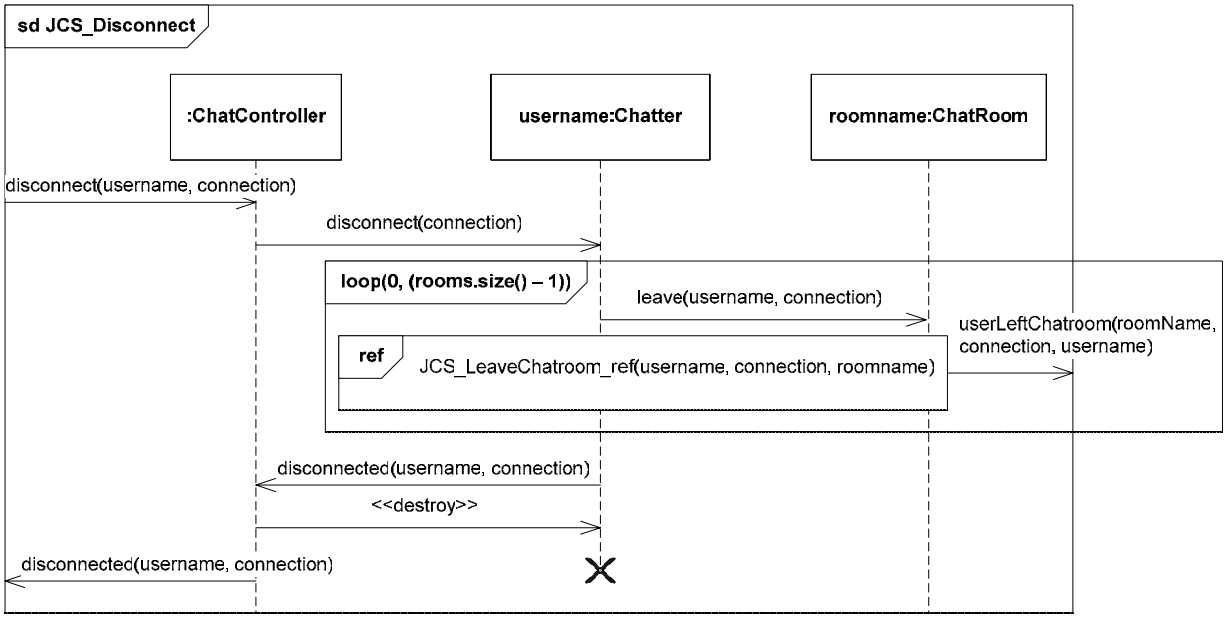
2.5.6. Disconnect og JCS_Disconnect

Når en bruker kobler seg fra serveren innebærer det at brukeren automatisk forlater alle rommene den er aktiv i, og blir deretter slettet fra serveren sin liste over aktive brukere.



Figur 2-16 sd Disconnect

Når det gjelder å koble seg fra serveren har jeg valgt bort å informere klienten om alle rommene den forlater. Dette fordi klienten i utgangspunktet kobler seg fra serveren og ikke prøver å forlate alle rommene.



Figur 2-17 JCS_Disconnect

2.6. Oppsummering

I dette kapittelet har jeg presentert spesifikasjonen for et pratesystem. Deler av modellene vil bli brukt i senere eksempler. Jeg tok denne gjennomgang av hele systemet så tidlig, for å gi deg som leser en bedre forståelse av de kommende eksemplene.

3. Modellbasert unntakshåndtering

Jeg vil her gi et overblikk over hva UML 2 og UML 2 Testing Profile (U2TP), gir av muligheter med tanke på modellbasert unntakshåndtering. Først vil jeg starte med å presentere hva UML 2 litteratur, og da spesifikk hva Rumbaugh et al. (2004) sier om unntak i sin andre utgave av UML Reference Manual.

3.1. Rumbaugh et al. sin definisjon på unntak

Da UML 2.0 spesifikasjonen (2004) ikke inneholder noen eksplisitt definisjon av unntak eller unntakshåndtering velger jeg å ta fram Rumbaugh et al. (2004) sin definisjon på unntak. Definisjonen er som følger:

> *"An indication of an unusual situation raised in response to behavioral faults by the underlying execution machinery or explicitly raised by an action. The occurrence of an exception aborts the normal flow of control …"* (Rumbaugh, 2004:338)

Definisjonen sier at unntak er et svar på en uvanlig situasjon som er et resultat av at noe har gått galt. Som Stroustrup har man her ikke valgt å si noe om hvem som skal håndtere unntaket, noe som gir oss ganske frie tøyler med tanke på asynkron unntakshåndtering. Det betyr at med denne definisjonen så kan vi i det mest ekstreme tilfellet la en bestemt livslinje ta seg av all unntakshåndtering, framfor å fordele dette over de faktiske klientene.

Et viktig poeng å ta med seg fra denne definisjonen er at det står det er en uvanlig situasjon. Det er altså en viss sammenheng mellom hyppigheten av en hendelse og det at det kan regnes som et unntak. Her blir det likevel snakk om kvalitative måleenheter, og det blir dermed opp til hver enkelt å definere seg hvor grensen for unntak går.

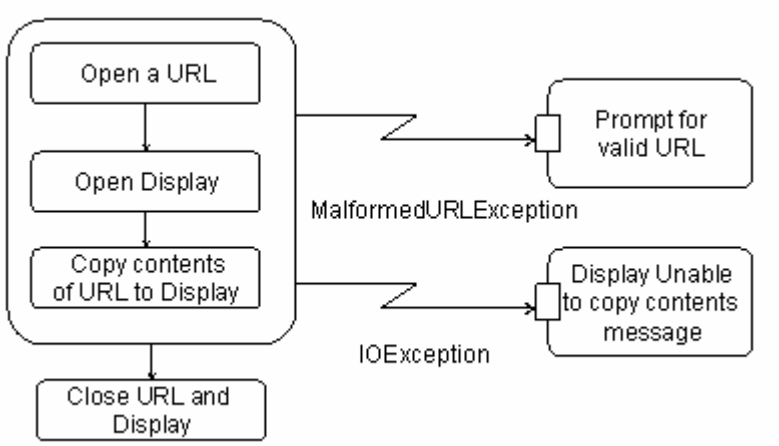
Videre er det viktig å gripe fatt i hva som står i siste setningen, nemlig at unntak avbryter normalflyten. Definisjonen sier altså at et program består av flere kontrollflyter, en normal (hva man ønsker å oppnå) og en eller flere unormale/eksepsjonelle (for å håndtere at noe går galt). Dette gir oss et utgangspunkt til å se på unntakshåndtering som noe som skal håndtere det som ikke hører hjemme i normalflyten, eller mer korrekt at det skal håndteres på en annen måte enn hva vanlige mekanismer for kontrollflyt tillater.

Noe Rumbaugh et al. (2004) sin definisjon sier lite om, er hva som skjer med normalflyten etter et unntak er håndtert. Det gies inntrykk av at det er en form for gjenopptagelse (eng.: *resumption*), altså det å kunne fortsette fra etter der hvor unntaket ble oppdaget, selv om de også sier at de baserer seg på C++ sine konsepter som ikke støtter gjenopptagelse.

3.2. Aktivitetsdiagrammer

Når det gjelder UML og unntakshåndtering har støtten for å modellere unntak fram til versjon 2 vært så godt som fraværende. I versjon 2 av UML har man nå fått utvidede mulighet, og man kan nå modellere unntakshåndtering i aktivitetsdiagrammer (Rumbaugh, 2004). Jeg vil her ta et raskt overblikk over mekanismene, og ikke gå i detalj da jeg i denne oppgaven ikke fokuserer på aktivitetsdiagrammer. [2]

For enkelhets skyld benytter jeg her eksempelet til Miller (2004) (figur 3-1). Eksempelet tar utgangspunkt i en nettleser, og hva som kan gjøres med denne og med tilhørende unntakshåndtering.



Figur 3-1 Eksempel på unntakshåndtering i aktivitetsdiagram

Figuren beskriver kort at man at har muligheten til å knytte håndtering av unntak opp mot bestemte unntak, som kan komme fra den omsluttede regionen. Hvis det hadde vært nestede aktiviteter inni regionen ville eventuelle unntak fra disse fulgt den statiske nesting ut på samme måte som om man hadde en kall stakk tilgjengelig. Man får gjennom nestingen tilgang til en form for kall stakk (Störrle, 2004).

Dette gjør at man får en strukturert måte å formidle unntak fra ulike nivåer på, som ikke ville vært mulig om man ikke kunne simulere en kall stakk. Man slipper altså å fortelle unntak hvor de skal finne en passende unntakshåndterer. Dette betyr derimot ikke at aktivitetsdiagrammer har noen kall stakk, for det har de ikke. Det betyr bare at den statiske nestingen gir noen av de samme mulighetene for aktivitetsdiagrammer, som en kall stakk gir i et programmeringsspråk.

Störrle (2004) tar også fram at det å flagge et unntak, i forhold til Petri Nets, kanskje kan sees på som det å returnere for tidlig fra en metode. Dette for å enkelt kunne mappe aktivitetsdiagrammer over til Petri Nets som mangler unntakshåndtering. Dette er fullt mulig, og hva man gjør når unntakshåndtering ikke er tilgjengelig. Problemet er bare at dette har lite med unntakshåndtering å gjøre da en viktig del er å skille normalflyten og

[2] For en grundigere analyse av unntakshåndtering i aktivitetsdiagrammer se Störrle(2004).

unntakshåndtering. I tillegg er det viktig å skape en løs kobling mellom det å sende og motta unntak, slik som i for eksempel Java. Dette skillet oppnåes ikke ved retur meldinger, spesielt fordi man da blander inn unntakshåndtering i normalflyten, men dette kommer jeg tilbake til i et senere kapittel.

Måten man beskriver unntakshåndtering i aktivitetsdiagrammer på har sine fordeler og ulemper. For det første så er det relativt oversiktelig, og rett fram måte å håndtere unntak på. Samtidig så kan dette bli veldig uoversiktlig, for det blir en voluminøs beskrivelse. Hvis det hadde vært 10-15 unntak, så ville det umiddelbart bli vanskelig å skille normalflyten fra unntakene.

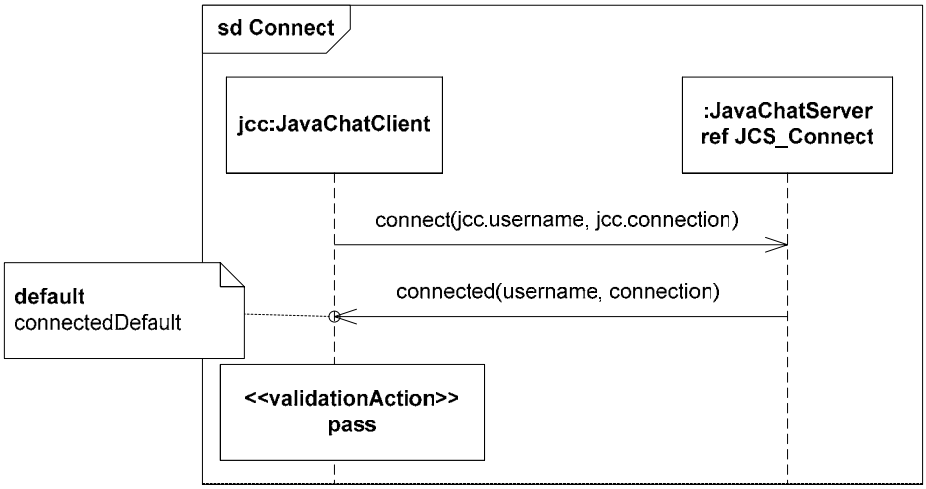
Med mindre det er en nesting av aktiviteter så blir det også fort én til én mapping mellom unntaket som kastes og hvordan det skal håndteres. I mange tilfeller så er man ikke interessert i hvordan unntaket håndteres, man ønsker kun å kaste ett unntak som så automatisk fanges opp av en unntakshåndterer.

Unntakshåndteringen i aktivitetsdiagrammer er et steg i riktig retning, men med tanke på kompakthet og løs kobling mellom det å kaste ett unntak, og det å håndtere ett unntak så er det mye å gå på. Störrle (2004) trekker spesielt fram problemer knyttet til unntak som kan kastes fra parallelle regioner, og de begrensede mulighetene til å kontrollere dette.

3.3. UML 2 Testing Profile

UML 2 Testing Profile (U2TP) er som navnet tilsier en profil av UML 2 spesielt laget for testing av programsystemer. U2TP tar hovedsakelig utgangspunkt i black-box testing. Dette innebærer at man kun ser systemet fra utsiden, og tester på om implementasjonen er i overensstemmelse med spesifikasjonen.

For å fange opp brudd på spesifikasjonen har man i U2TP definert seg noe man kaller for *defaults*. Defaults kan blant annet knyttes opp mot hendelser (eng.: *events*) for å kunne håndtere andre meldinger enn den som er spesifisert (se figur 3-2).

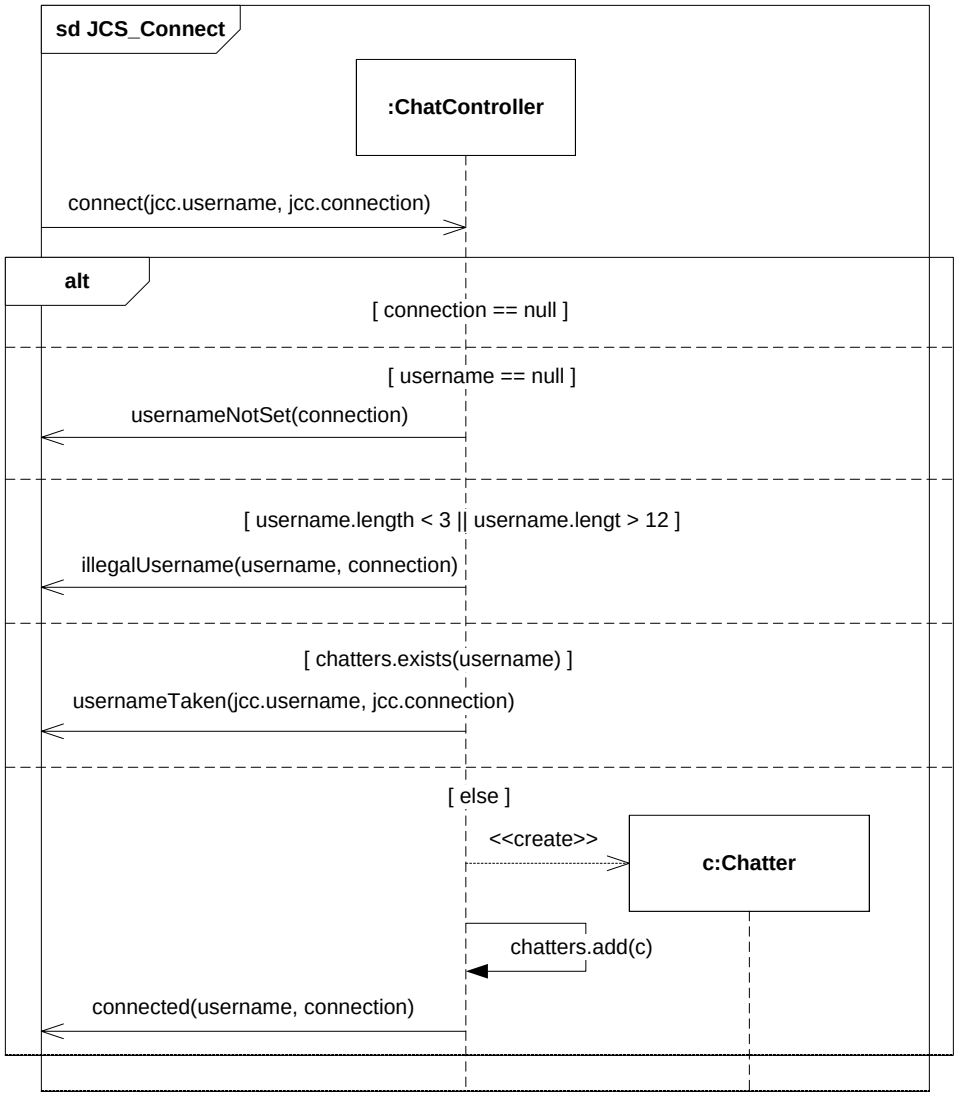


Figur 3-2 Eksempel på default knyttet opp mot en melding

I figur 3-2 er det spesifisert at etter man har sendt en connect melding, så skal man motta en connected melding. Hvis det kommer en annen melding enn connected så håndteres dette av en default connectedDefault.

Defaults er veldig sterkt knyttet opp mot unntakshåndtering. I unntakshåndtering ønsker vi nettopp å kunne håndtere uventede hendelser. Som jeg nevnte så tar U2TP utgangspunkt i black-box testing. Dette gjør at man ikke modellerer bruddene på spesifikasjonen, noe man må gjøre i unntakshåndtering, men kun håndtering av bruddene. I unntakshåndtering må man ha kontroll over bruddene, og det får man ikke uten å modellere dem.

Sett at man for sd Connect skulle ha modellert alle mulige tilfeller av meldinger som kunne komme ut, da ville JCS_Connect sett ut som vist i figur 3-3.



Figur 3-3 JCS_Connect

Hvis man sammenligner figur 3-3 og figur 3-2 ser man at det er problemer med gate matching. Det sendes ut flere mulige meldinger enn hva som blir mottatt. Dette problemet trenger ikke U2TP å bry seg med nettopp fordi U2TP baserer seg på black-box testing, og testing på om implementasjonen overholder spesifikasjonen. Dette betyr at man ikke modellerer sending av unntaksmeldinger. For unntakshåndtering så må man derimot være i stand til å både ta imot unntak og å informere om unntak/sende unntaksmeldinger, og samtidig overholde gate matching.

En foreløpig vurdering av U2TP vil være at testing kan ansees som et subset av unntakshåndtering. Dette vil jeg begrunne med at både testing og unntakshåndtering er opptatt av å håndtere noe som bryter med en gitt spesifikasjon. Grunnen til at U2TP kan ansees som et subset av unntakshåndtering er at U2TP mangler muligheter for å eksplitt informere om unntakssituasjoner (sende uventede meldinger), men U2TP har mekanismer for å ta imot unntak.

Selv om U2TP kan ansees som et subset av unntakshåndtering er ikke det ensbetydende med at U2TP kan utvides til bruk for unntakshåndtering. Gjennom å slippe å tenke på gate matching har U2TP et utgangspunkt som er vanskelig å bygge videre på, da man ved sending av unntak må ta hensyn til gate matching.

3.4. Forskjellen på modellering og programmering

I problemstillingen sa jeg at jeg skulle se på om det var noen forskjell mellom hvordan unntakshåndtering bør brukes i modellering og programmering. Dette spørsmålet besvares egentlig best ved å se på hva forskjellen mellom modellering og programmering er, da disse forskjellene vil legge føringer på hva man kan oppnå ved bruk av unntakshåndtering.

Modellering og programmering er to sider av samme sak, men likevel ganske forskjellige. Jeg vil her sette modellering og programmering opp mot hverandre for å se på hvor disse står i forhold til hverandre med tanke på unntakshåndtering. For å relatere dette til denne oppgaven, så vil jeg her se på forholdet mellom unntak i sekvensdiagrammer og Java exceptions.

Et viktig spørsmål før jeg begynner å gå inn på unntakshåndtering i sekvensdiagrammer, vil være å ta en overordnet sammenligning og se på hva Java exceptions tilbyr og trenger, for så å se hva av dette sekvensdiagrammer har.

Java exceptions tilbyr i utgangspunktet to ting:

- Endring av kontrollflyt
- Løs kobling mellom sending og mottak av exceptions.

Endring av kontrollflyt innebærer at man i Java kan ha en normalflyt som man følger, og ved brudd på normalflyten så trer unntakshåndteringen inn. Dette innebærer at man slipper å blande sammen normalflyt og unntakshåndtering. I Java lages dette skillet ved hjelp av en try/catch konstruksjon.

Eksempel:

```
try{
  //normalflyten
}
catch(SomeException e){
  //håndtere spesifikt unntak
}
catch(Exception e){
  //håndtere generelt unntak
}
```

Denne try/catch konstruksjonen gjør at man får delt opp normalflyten og unntakshåndteringen. Dette gjør koden lettere å lese, og lettere å skrive.

Når det kommer til løs kobling mellom sending og mottak av unntak går dette på at man hos serveren ikke behøver å tenke på hvilket objekt som skal motta unntaket. Serveren kaster kun unntaket. Som jeg beskrev tidligere behøver ikke serveren å kjenne mottaker, fordi man har en dynamisk kall stakk tilgjengelig. På denne stakken så skal det finnes en som kan håndtere unntaket, og man finner en passende catch ved å poppe seg langt nok tilbake på denne stakken.

I forhold til sekvensdiagrammer er endring av kontrollflyt noe som mangler. Man har konstruksjoner som *alt* og *opt*, men dette representerer ikke noen endring av kontrollflyt. *Alt* og *opt* er kun valg på lik linje med en *if* test. Det som trengs er noe som kan skape et visuelt skille på linje med en try/catch konstruksjon.

Når det kommer til det å sørge for en løs kobling mellom sending og mottak av unntak, så er det naturlig nok helt fraværende. Dette fordi sekvensdiagrammer mangler muligheter for unntakshåndtering. Som jeg beskrev tidligere muliggjøres denne løse koblingen i Java ved hjelp av den dynamiske kall stakken. I forhold til sekvensdiagrammer er dette en stor utfordring da sekvensdiagrammer ikke har noen dynamisk kall stakk. Mangelen på en dynamisk kall stakk gjør at den løse koblingen mellom sending og mottak av unntak i Java blir vanskelig å oppnå.

På den andre siden så brukes Java exceptions og sekvensdiagrammer gjerne på to forskjellige områder. Java exceptions kan kun kastes og fanges i samme tråd (eng. *thread*) (Lindholm et al., 1999). Dette er fordi at når en tråd blir opprettet får den sin egen JVM stakk. På denne JVM stakken legger tråden til en ny stack frame for hvert metodekall i tråden. En stack frame har blant annet i oppgave å formidle unntak, i tillegg til å holde på lokale variable, del resultater osv.. Det som gjør at man ikke kan kaste unntak mellom tråder, er nemlig det at man ikke har anledning til å referere til en annen tråds stack frames og har dermed heller ikke lov til å kaste unntak til en annen tråd. Hver tråd har altså sin helt private JVM stakk.

Sekvensdiagrammer på sin side, beskriver ofte asynkron meldingsutveksling, og har ikke noen kall stakk som kan brukes ettersom man har gjerne jobber over flere tråder. Dette gjør at man må bruke andre mekanismer for å automatisk få sendt unntak til rett mottaker.

3.5. Oppsummering

Jeg har her sett på hva UML litteratur sier om unntak og unntakshåndtering. Videre så jeg på aktivitetsdiagrammer og hvilke styrker og svakheter som var ved unntakshåndteringen som der var innført. Styrker og svakheter i forhold til kompakthet og løs kobling mellom sending og mottak/håndtering av unntak. Jeg viste videre at U2TP gav gode muligheter med tanke på å ta imot uventede meldinger, men at de samtidig ikke hadde noen løsning på hvordan disse meldingene ble sendt.

Jeg så deretter på sekvensdiagrammer og Java og påpekte at det er grunnleggende forskjeller mellom disse med tanke på kall stakken. Hver tråd i et Java program har sin egen kall stakk, mens sekvensdiagrammer ikke har noe tilsvarende. Denne forskjellen gjør at man må se seg om etter andre mekanismer til å formidle unntak i sekvensdiagrammer

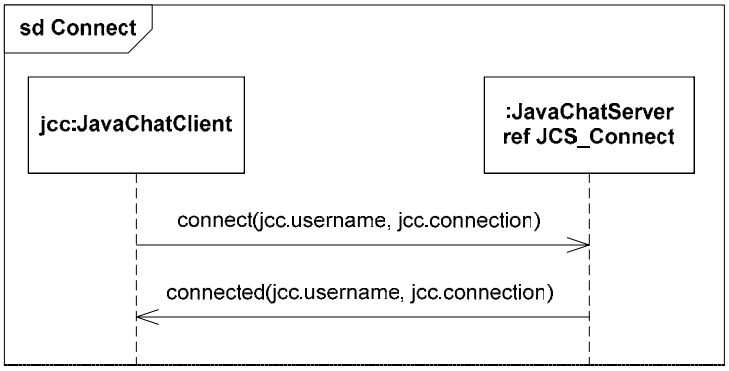
Unntakshåndtering handler likevel generelt om det samme, enten det er programmering eller modellering. Det handler om å håndtere uventet oppførsel. Forskjellen ligger i de tekniske aspektene ved implementasjonen, og ikke i målsetningene/bruksområdene for mekanismene.

4. Unntakshåndtering i sekvensdiagrammer

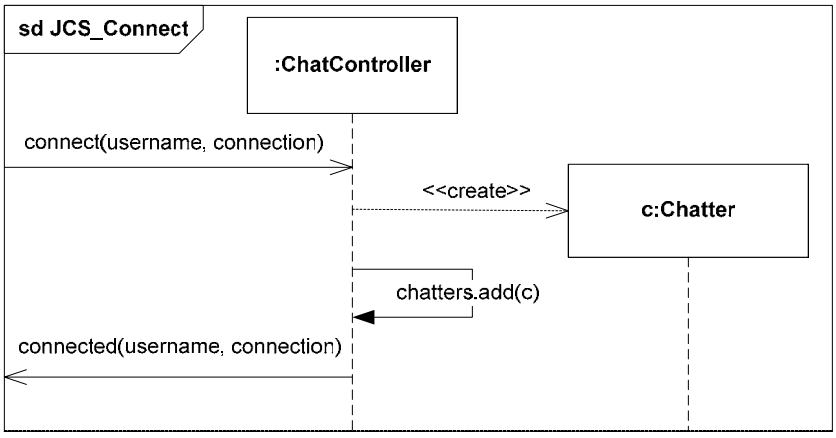
Før jeg kan gå over på neste del av problemstillingen, nemlig forslag til mekanismer for unntakshåndtering, vil jeg først presentere motivasjonen og målene jeg har for disse mekanismene. Jeg vil begynne med et lite eksempel for å vise at mekanismene i UML 2.0 sine sekvensdiagrammer er utilstrekkelige for å håndtere unntak. Deretter vil jeg presentere de målene jeg har satt for mekanismer til unntakshåndtering, og hvilke utfordringer disse målene gir.

4.1. Unntakshåndtering ved bruk av dagens mekanismer

For å få fram motivasjonen for å innføre mekanismer for unntakshåndtering vil jeg her ta fram det relativt enkle case'et med å koble seg opp mot prateserveren. Normalflyten for det å koble seg opp mot prateserveren er vist i figur 4-1 og figur 4-2.



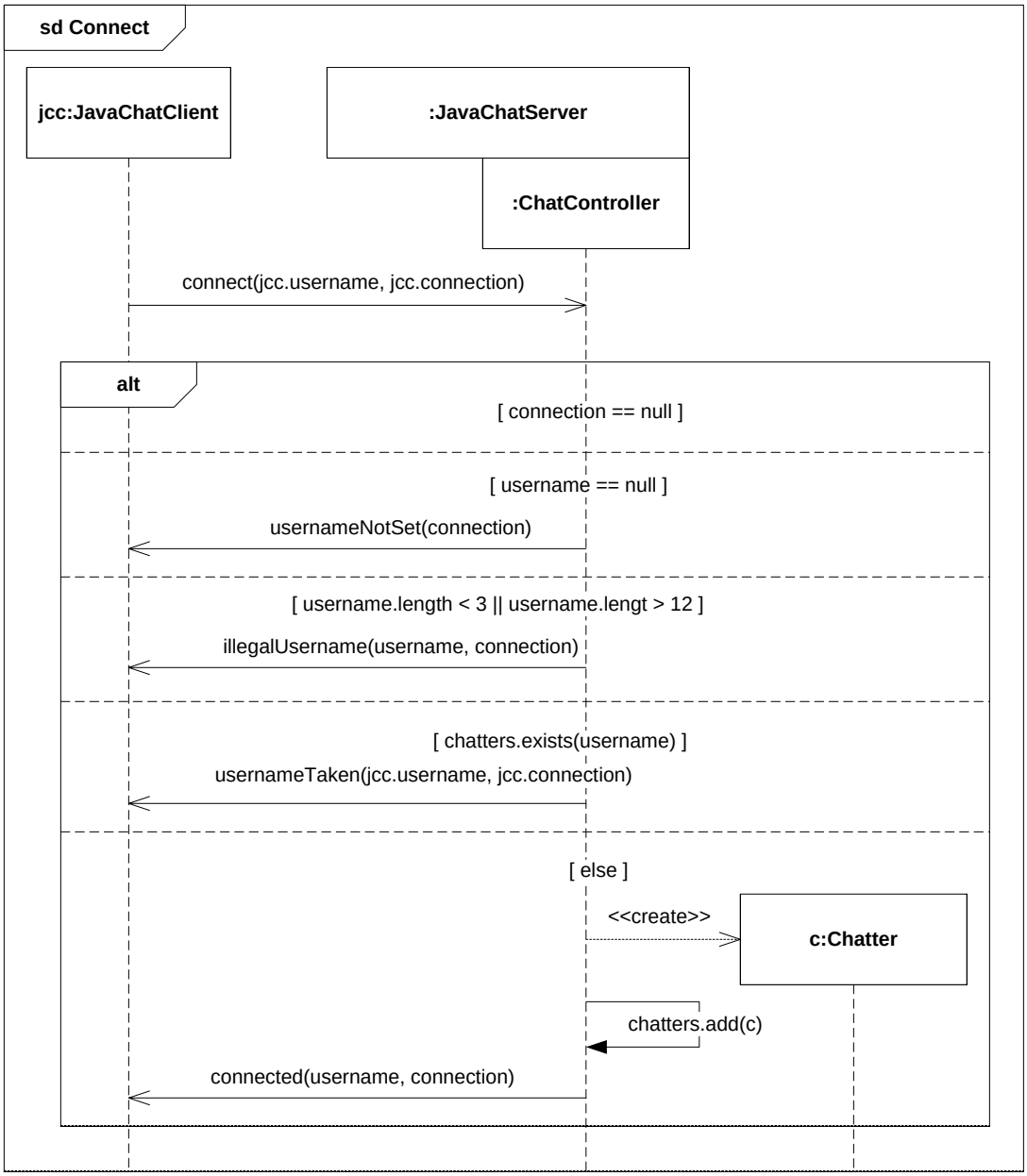
Figur 4-1 sd Connect



Figur 4-2 sd JCS_Connect

For at denne spesifikasjonen skal kunne fungere korrekt må en beskrive en del unntak også. Man må for eksempel sjekke at det ikke er sendt med null verdier, deretter så må det sjekkes at det er lovlig brukernavn og at brukernavnet er ledig. Ikke noe av det nevnte er så langt tatt høyde for.

Figur 4-3 viser hvordan de nevnte unntakene kan håndteres i UML 2.0 uten spesialiserte mekanismer for unntakshåndtering.



Figur 4-3 Håndtere unntak ved bruk av UML 2.0 sine mekanismer.

Som figur 4-3 viser, er potensielle unntak håndtert ved hjelp av en *alt* konstruksjon. Det er ikke lengre entydig hva som er normalflyt og hva som er unntak. Vi mangler altså et klart visuelt skille mellom normalflyten og unntakene. Dette er blant annet hva jeg søker å løse ved å innføre mekanismer for unntakshåndtering.

4.2. Overordnede mål for mekanismer til unntakshåndtering

Som vist i kapittel 4.1. er det ved dagens mekanismer umulig å skille det vesentlige fra det uvesentlige på en enkel, kompakt og godt oversiktelig måte. I prioritert rekkefølge anser jeg de følgende målene som viktigst for mekanismer til unntakshåndtering:

1. Skille normalflyten fra unntakene/unntakshåndteringen.
2. Kompakte mekanismer for å håndtere unntak.
3. God oversikt over den nye kontrollflyten unntakshåndtering gir.

Når det gjelder å skille normalflyten fra unntakshåndtering er det helt sentralt. Terry Winograd (1979) beskriver denne utfordringen på følgende måte:

> *"If there are exceptional cases (e.g. when the storage allocator fails to find a sufficient block), these need to be described, but in a secondary place. This basic description of an object cannot be cluttered up with all of the details needed for handling the contingencies."*

Uten å ha mulighet til å skille normalflyten fra unntakene så sitter vi igjen med det som allerede finnes og ender da opp med, som Winograd (1979) påpeker, å fylle opp normalflyten med unntakshåndtering. Figur 4-3 viser også godt dette poenget. I denne modellen er det for tidkrevende å finne fram til det vesentlige, til tross for at det er et veldig lite eksempel. Det er altså helt vesentlig å få til et visuelt skille mellom unntak, og normalflyt.

Det at mekanismene for å håndtere unntak må være kompakte går det på at man i utgangspunktet ikke er interessert i unntak. I utgangspunktet så er man kun interessert i å se på normalflyten. Unntak er noe som skal ligge i bakgrunnen og sørge for at normalflyten kan gå sin gang. Det er altså helt sentralt med unntakshåndtering, men mekanismene må være så kompakte at de ikke overskygger normalflyten.

Selv om unntakshåndter gjør det enklere å sette seg inn i normalflyten, medfører det en betydelig mer komplisert kontrollflyt. Man går fra å blande unntakshåndtering med normalflyten, til å legge dette i bakgrunnen. Dette gjør at man får en ekstra dimensjon i kontrollflyten som gjør at man ikke nødvendigvis enkelt kan følge et program fra A til Å, men man risikerer å bli sendt fram og tilbake mellom unntakshåndtering og normalflyt.

En stor utfordring her blir å håndtere mulige kritiske regioner som unntakshåndtering i et asynkront og parallelt miljø vil medføre. Det vil ikke være opp til mekanismene for å unntakshåndtering å håndtere kritiske regioner, men det vil være opp til den enkelte designer. Å gjøre denne jobben enklere vil være viktig for å få korrekte programmer. Mekanismene for unntakshåndtering bør derfor tilby tilstrekkelig oversikt over kontrollflyten til at disse tilfellene enkelt kan avdekkes og håndteres av designeren(e).

4.3. Ulike scenarier for unntakshåndtering

Når det gjelder unntakshåndtering er det flere scenarier mekanismene må kunne håndtere. Tre viktige scenarier er følgende:

1. Evaluering/testing av betingelser
2. Overvåking av kjøring
3. Motta uventet melding

Den store forskjellen på disse 3 scenariene er hvem som utfører hva. Når det gjelder første punktet, evaluering av betingelse er dette noe serveren (den som mottar en melding) gjør. En server mottar en melding fra en klient (den som sendte meldingen), serveren evaluerer meldingen, hvis betingelsen slår til så er det en unntakssituasjon som må håndteres. I henhold til Goodenough (1975) kan ikke den som oppdager unntaket vite hva som skal gjøres, og må derfor informere en annen livslinje om at et unntak har oppstått.

Livslinje \ Scenario	**Testing**	**Overvåking**	**Uventet melding**
Klient	X	X	X
Server	X		X

Tabell 4-1 Hvem som er aktuell for de ulike scenariene

Når det gjelder overvåking av kjøring er dette noe en klient gjør. En livslinje kan være interessert i å se om en mengde handlinger går i henhold til spesifikasjonen. Hvis noe går galt ønsker man å kunne håndtere dette, og eventuelt fortsette der normalflyten slapp. I andre tilfeller oppdager en så alvorlige unntak at man må gi opp å utføre handlingen.

Det å teste på betingelser og det å overvåke kjøring henger tett sammen. På et lavt nok nivå så må en betingelse testes før et unntak kan sendes. Derimot så har man ikke alltid full oversikt over unntakshåndtering, kanskje noen andre har laget en komponent du bruker. Dette gjør at verken klienten eller serveren i utgangspunktet er interessert i den andre. Serveren på sin side er interessert i å sende eventuelle unntak, og ikke hvem de skal sendes til. Klienten på den andre siden er interessert i å kunne motta eventuelle unntak, uavhengig av hvor de kommer fra.

Når det gjelder uventet melding er dette noe som er aktuelt for både klient og server, nettopp fordi den er uventet. Med uventet melding så menes en melding som ikke er tatt med i spesifikasjonen av normalflyten. Et spørsmål her er hvorfor dette er relevant, da det i dette tilfellet vil være et resultat av en ufullstendig spesifikasjon. Faktum er at det ikke er lett å få med seg alt, og derfor er dette punktet veldig sentralt i forhold til å ha et robust system for unntakshåndtering.

Man kunne også tenke seg at alle unntak var uventede meldinger. Dette innebærer at unntak kan inntreffe når som helst, og hvor som helst. I forhold til det å skape en svak kobling mellom det å sende og motta unntak, er dette bra. Det gir oss heller ikke noe behov for å ligge og vente på eventuelle unntak. På den andre siden så er ofte unntak et resultat av at man har sendt en forespørsel til en server. Dette på sin side gir en ganske streng form for hvor og når unntak kan oppstå. Unntak kan da kun oppstå som resultat av et kall til en server.

Likevel så er det naturlig å tenke på unntak som uventede meldinger. Dette gjør at vi kan flette sammen de 3 scenariene jeg spesifiserte. Det å teste på betingelse kan resultere i at man sender ett unntak. Sett med mottaker sine øyne representerer dette en uventet melding. Det å overvåke en kjøring går på å være klar til å ta imot eventuelle uventede meldinger.

Det å sende og motta uventede meldinger i sekvensdiagrammer er ikke helt rett fram, ettersom man i sekvensdiagrammer spesifiserer tracer. En trace beskriver en kjøring, og i forhold til en trace kan det ikke skje noe uventet fordi en trace er endelig. Dette gjør at hva som i sekvensdiagrammer egentlig skal være uventet, må være mulig å få utledet strukturerte tracer for. Denne og andre utfordringer kommer jeg tilbake til i neste kapittel.

4.4. Utfordringer knyttet til unntakshåndtering og sekvensdiagrammer

For å kunne håndtere de ulike scenariene og nå målene jeg skisserte i kapittel 4.3 må sekvensdiagrammene utvides. Det er flere store utfordringer å komme rundt. De mest aktuelle problemstillingene er:

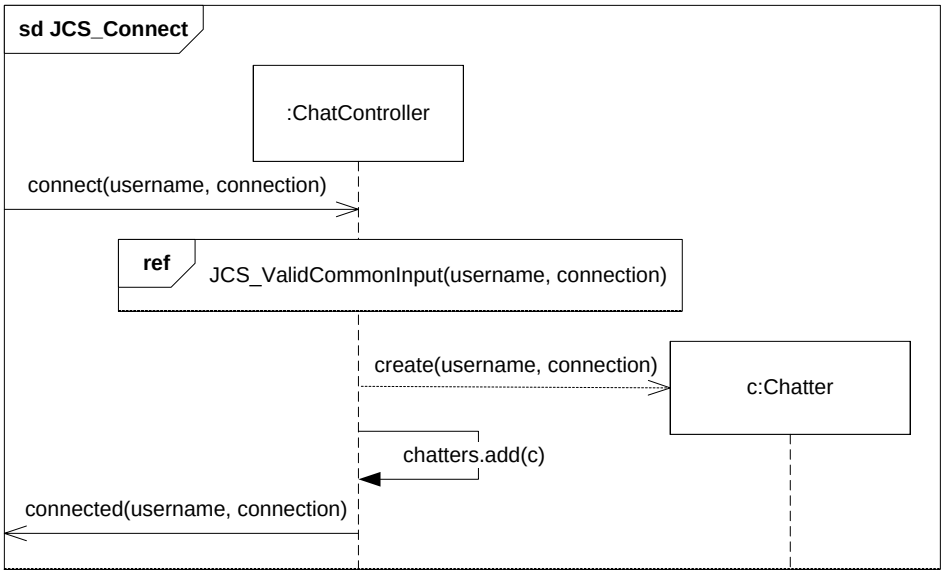
- Visuelt skille mellom normalflyt og unntakshåndtering
- Retur kontra terminering
- Sekvensdiagrammer har ikke har noen dynamisk kall stakk (eng.: *call-stack*)
    - Utfordringer med tanke på å få en løs kobling mellom sending og mottak av unntak

Jeg kommer her til å utdype de nevnte problemstillingene, og skissere forslag til løsninger. Forslagene til løsning kommer så i neste kapittel til å bli videre utdypet og satt sammen til et endelig forslag til mekanismer for unntakshåndtering.

4.4.1. Skille normalflyt og unntakshåndtering

Utfordringen ved å skille normalflyt og unntakshåndtering handler egentlig om gate matching. Når vi søker å flytte noe ut av normalflyten, så resulterer det i at de tilhørende gatene også må flyttes.

Sett at man ønsker å flytte all unntakshåndtering ut av normalflyten fra figur 4-3. Da kunne man tenke seg å gjøre dette ved hjelp av en form for referanse, og vil gi noe tilsvarende hva som er vist figur 4-4.

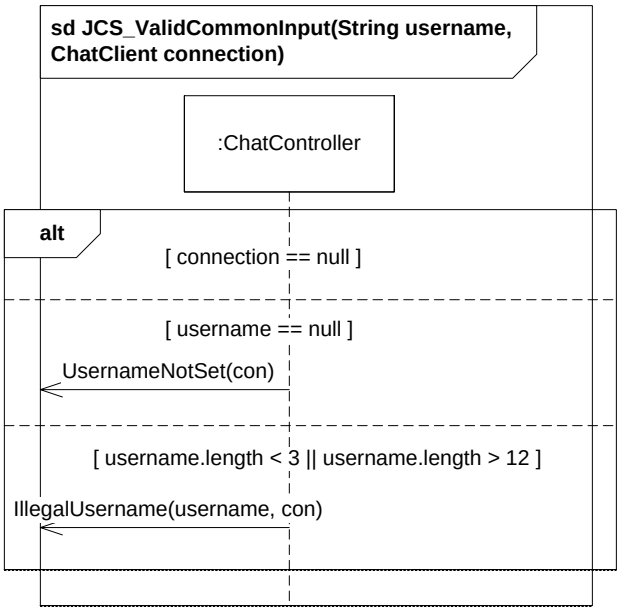


Figur 4-4 Fjerne unntakshåndtering ved hjelp av referanse

Problemet med dette er to ting. Først og fremst så får man problemer med tanke på at *alt* konstruksjonen ikke lengre omfatter normalflyten, og dermed så avsluttes heller ikke tracen hvis en av blokkene til alt'en slår til. Tracen vil bli som følger:

Connect = <!connect, ?connect> seq JCS_ValidCommonInput seq <!chatters.add, ?chatters.add, !connected, ?connected>

Den tracen er ikke hva vi ønsker å si.



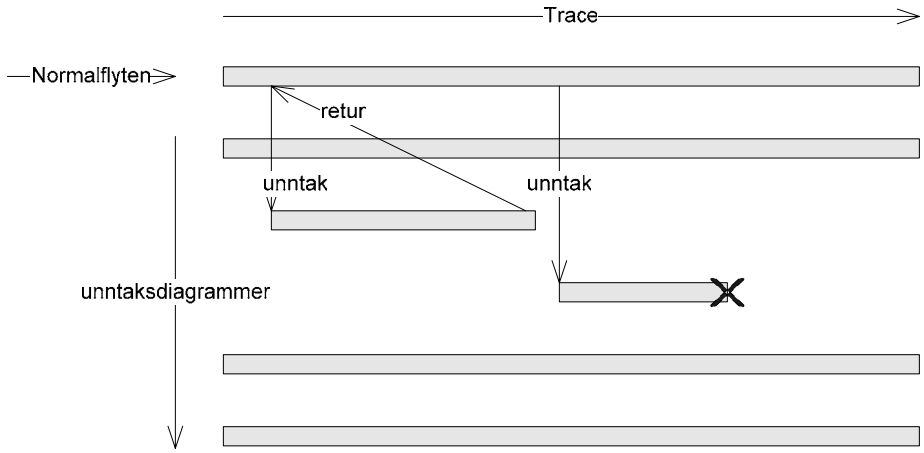
Figur 4-5 JCS_ValidCommonInput

Dernest så er ikke de utgående gatene fra JCS_ValidCommonInput med i JCS_Connect, og hvis de blir tatt med så er man mer eller mindre tilbake til utgangspunktet med blanding av unntakshåndtering og normalflyt.

For å kunne fjerne unntakshåndtering fra normalflyten trenger man altså en konstruksjon som gir oss muligheten til å komme rundt problemet med gates.

Til å løse utfordringen med gate matching må man innføre flere nivåer med gates. Jeg lar sekvensdiagrammer gå over flere logiske plan, hvor hvert plan har sine egne gates. Logiske plan betyr at dette er en måte å strukturer tracer i sekvensdiagrammer på, og vil ikke nødvendigvis være direkte overførbart til den implementerte løsningen.

Man kan se for seg mengden av plan som et hus med en hovedetasje og et ubegrenset antall underetasjer. Hovedetasjen er forbunnet med underetasjene gjennom en mengde heiser. Hovedetasjen representerer normalflyten, mens underetasjene representerer deler av tracen som trenger å sende meldinger ut noen andre porter. Hvert plan har altså sine egne unike gates.



Figur 4-6 Eksempel på kjøring over flere plan

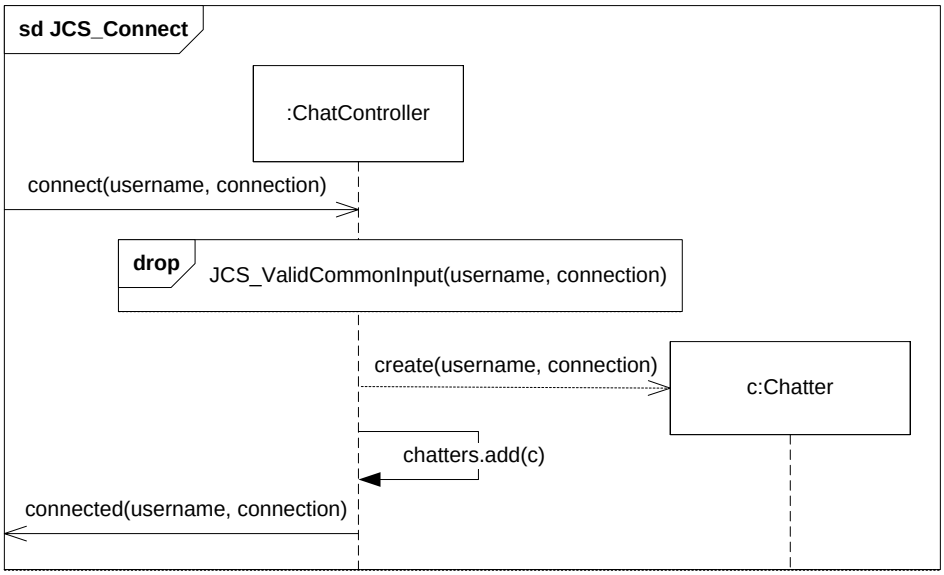
I Figur 4-6 har jeg angitt at man både skal kunne returnere til normalflyten, og kunne avslutte tracen på et lavere nivå. Dette med å avslutte tracen på et lavere nivå byr på visse utfordringer med tanke på å trace semantikken. Mer om retur og terminering i kapittel 4.4.2.

Planløsningen er likevel ikke noen løsning på hele utfordringen med unntakshåndtering. Vi kan på denne måten komme rundt utfordringen med sammenblanding av unntakshåndtering og normalflyt, men det er fremdeles en sterk kobling mellom sending og mottak av unntak. I det så ligger det at tracen fortsetter på vanlig måte, men kun på et lavere nivå. Eksempel på dette finnes i figur 4-7 og figur 4-8. Vi mangler altså et viktig element, sett i forhold til Java exceptions, med å separere sending og mottak av unntak.

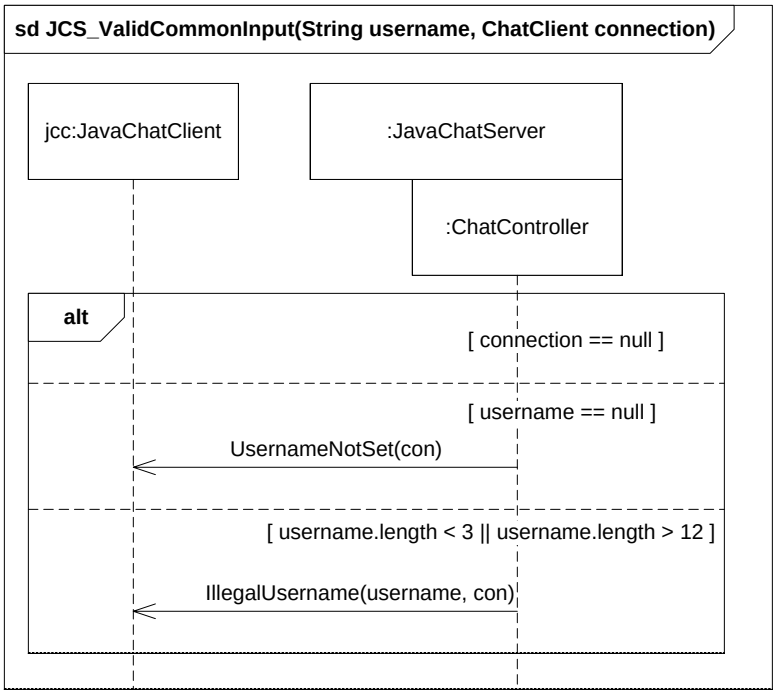
4.4.2. Retur kontra Terminering

I kapittel 4.4.3 ser jeg videre på utfordringene ved å skille sending og mottak av unntak. Jeg går nå tilbake til problematikken med retur og terminering.

Når det gjelder planløsningen er den, som nevnt, tenkt brukt for å komme rundt problemet med gate matching. For å få til dette innfører jeg en ny operator *drop*, som flytter kontrollflyten til en nytt plan. Dette innebærer at man ved å flytte JCS_ValidCommonInput til et annet nivå vil komme rundt problematikken med gates.



Figur 4-7 Eksempel på drop



Figur 4-8 ValidCommonInput

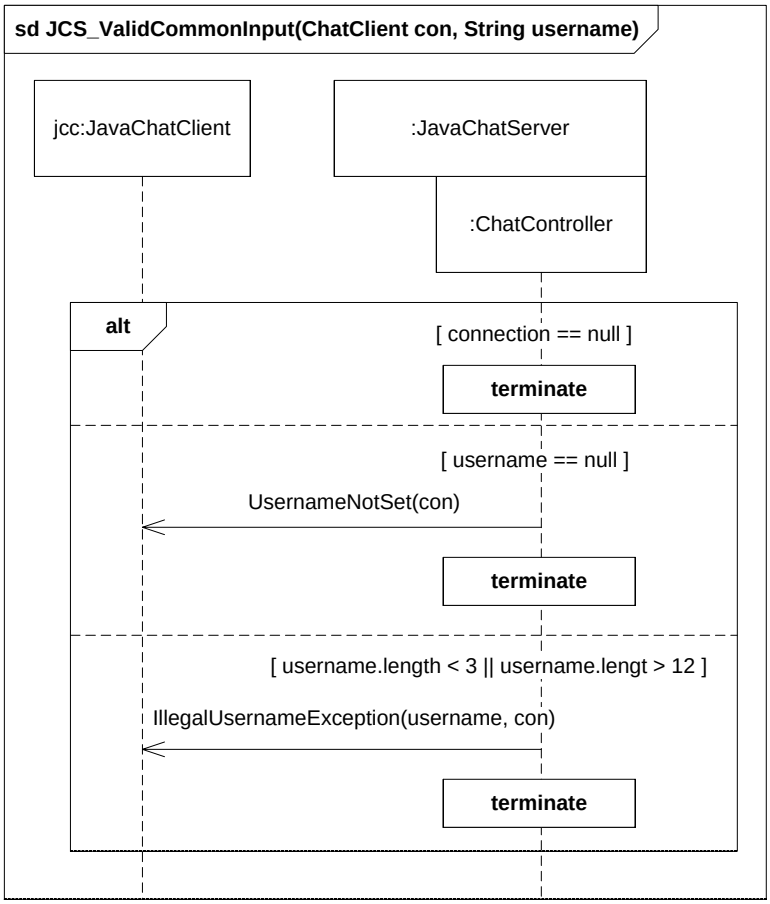
Ved hjelp av drop har jeg nå kommet meg rundt problemet med gate matching, men jeg har ikke løst utfordringen ved å få terminert en trace etter alt konstruksjonen. Sånn som det er modellert i figur 4-7 og figur 4-8 vil lovlige tracer bli som følger:

Connect = <!connect, ?connect> seq JCS_ValidCommonInput seq <!chatters.add,
?chatters.add, !connected, ?connected>

I forhold til at man i dette tilfellet ønsker at tracene skal termineres etter alt'en blir ikke dette rett. Det jeg egentlig ønsker å si at *drop* er består av litt mer, nemlig at en drop resulterer i at tracen blir som følgende:

Connect = <!connect, ?connect> seq (drop JCS_ValidCommonInput opt (<!chatters.add,
?chatters.add, !connected, ?connected>))

Dette betyr at drop legger til en opt i forhold til resten av tracen. Denne opt'en evaluerer automatisk resultatet av en drop. Ved retur slår opt'en til, og ved terminate slår den ikke til. På denne måten får man fram at drop kan resultere i et tracen avbrytes. På modellnivå kan man angi terminering som vist i figur 4-9. Retur blir tilsvarende, bare det at man bytter ut *terminate* med *return*.



Figur 4-9 Eksempel på bruk av terminate

4.4.3. Dynamikk

Som jeg beskrev i kapittel 3.4 er det grunnleggende forskjeller mellom sekvensdiagrammer og Java exceptions. Sekvensdiagrammer har ingen dynamisk kall stakk noe som gjør at man ikke sende ett unntak uten å måtte tenke på hvem som fysisk tar imot dette. Dette nettopp fordi man ikke har noen kall stakk å lete seg gjennom. Mangelen på en kall stakk gjør det problematisk å få en løs kobling mellom sending og mottak av unntak. Løs kobling i den forstand at man ikke skal behøve å spesifisere mottakeren.

Da sekvensdiagrammer mangler en kall stakk har de i utgangspunkt lite til felles med Java exceptions, men der i mot mer til felles med avbruddshåndtering (eng.: *interrupt handling*). Avbrudd fungerer på den måten at man har parallelle prosesser og ved gitte situasjoner så sender man et avbrudd til en bestemt mottaker. Ettersom det er parallelle prosesser har man heller ikke her noen dynamisk kall stakk.

Spørsmålet er om avbrudd og avbruddshåndtering er et bedre utgangspunkt, enn det å se på Java exceptions. Avbrudd har mye Java exceptions ikke har. Avbrudd er 100% asynkront, og kan dermed sendes og mottaes når som helst. I forhold til løs kobling

mellom avsender og mottaker kan det være et problem med avbrudd, og det er at avbrudd har en gitt mottaker. Det blir ikke noen dynamikk i forhold til sender og mottaker, sånn som man får ved Java exceptions. I forhold til sekvensdiagrammer er kanskje ikke dette så dumt, da disse er statiske. På den andre siden så vil den typen unntakshåndtering ikke gi så veldig stor gevinst med tanke det å formidle unntak mellom server og klient.

Ideelt sett skulle man kunne slippe unntak hvor som helst uten tanke på hvem som håndterer disse. Man skulle i tillegg ha mulighet til å motta unntak når som helst, uten å tenke på avsender – selv om noen naturlig nok må sende unntakene.

Når det gjelder det å ta imot unntak når som helst vil det kreve at hver livslinje i utgangspunktet har sin egen unntakshåndterer. Disse unntakshåndtererene overvåker så en trace for et gitt sekvensdiagram, eller deler av denne tracen. Det neste spørsmålet da blir når man kan ta imot unntak. Ettersom det her er snakk om asynkron meldingsutveksling behøver det ikke å være så strengt når de kommer fram. Dette fordi det er asynkront, og at det da ikke finnes noen garanti for når meldinger kan mottaes. Med dette utgangspunkt kan man egentlig velge å ta imot unntaket når man selv ønsker, og på den måten skape en viss fleksibilitet med tanke på å unngå å sette seg selv i vanskelige situasjoner.

Et annet problem med mottak av unntaksmeldinger, er hvilke unntak som er svar på hvilke melding. Sett at man ønsker å angi at en livslinje vil overvåke et begrenset område med meldingsutveksling for å kunne håndtere eventuelle unntak knyttet til disse. Hvordan vet man da at de eventuelle unntakene er svar på en gitt melding? I utgangspunktet så kan de være svar på noe helt annet.

For at dette skal kunne løses så må det altså være en viss sammenheng mellom meldingen som sendes og unntakene som kan mottaes. Man må ha en spesifikasjon som angir hvilke unntaksmeldinger som kan være resultat fra en melding. Med tanke på å bruke U2TP sine defaults for å ta imot uventede meldinger byr dette på et problem. Sett i forhold til en implementasjon kan man fort tenke seg at uvettig bruk av default resulterer i at en default fanger opp unntak som var ment på en annen default.

4.5. Oppsummering

I dette kapittelet har jeg vist hva mangelen på mekanismer for unntakshåndtering medfører. Jeg har presentert målene for mekanismene til unntakshåndtering og hvilke scenarier mekanismene må håndtere.

Viktige mål gikk på å skape et visuelt skille mellom unntakshåndtering og normalflyten, og å være i stand til å håndtere den ekstra kompliserte kontrollflyten unntakshåndtering medfører. Videre så presenterte jeg det at mekanismene må være i stand til både å overvåke sekvensdiagram eller deler av sekvensdiagram, sende unntak og motta uventede meldinger (unntak).

Jeg så deretter på problemstillinger knyttet til det å skille normalflyt og unntakshåndtering. Så at dette kunne løses ved en planløsning. Videre tok jeg for meg hvordan man kunne flytte seg mellom de ulike planene ved hjelp av retur og terminering og hvilke utfordringer dette skapte for tracen. Til slutt tok jeg for meg utfordringer knyttet til løs kobling mellom sending og mottak av unntak. Vurderte om sekvensdiagrammer ville ha mer å hente fra avbruddshåndtering enn fra Java exceptions, og hvilke utfordringer man kunne støte på ved bruk av en avbruddslignende unntakshåndtering.

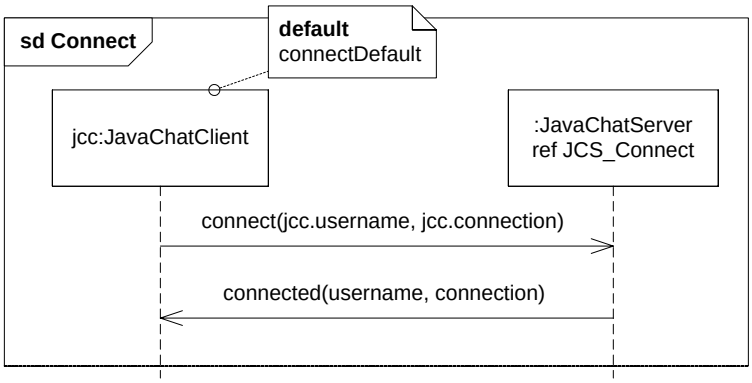
5. Forslag til mekanismer for unntakshåndtering i sekvensdiagrammer

Jeg vil her presentere et forslag til mekanismer for unntakshåndtering i sekvensdiagrammer. Først vil jeg presentere noen ulike eksempler som benytter seg av mekanismene, for til slutt å se på den presise semantikken til mekanismene.

5.1. Planløsning

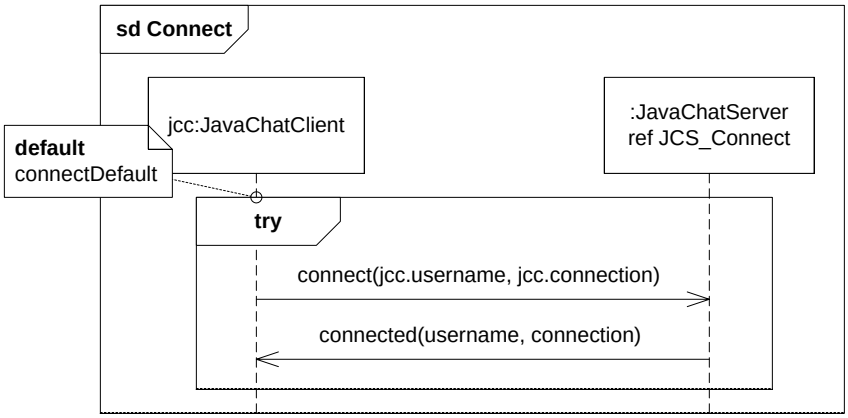
Forslaget jeg her vil her presentere baserer seg på at sekvensdiagrammer går over flere logiske plan. Logiske plan betyr at dette er en måte å strukturer tracer i sekvensdiagrammer på, og vil ikke være direkte overførbart til den implementerte løsningen. Grunnen til at jeg ønsker å innføre flere logiske plan, er for å komme rundt utfordringen med gate matching og for å kunne skape et visuelt skille mellom normalflyt og unntaksflyt.

Jeg vil her bruke eksempelet med det å koble seg til prateserveren. Det er et lite og enkelt eksempel som kan presentere teknikkene. Modellen i figur 5-1 viser normalflyten for hvordan en klient kobler seg opp mot prateserveren. Jeg har her lagt til en default, jccConnect. Default'en er koblet til objektet jcc. Dette betyr at default'en gjelder jcc i hele sd Connect.



Figur 5-1

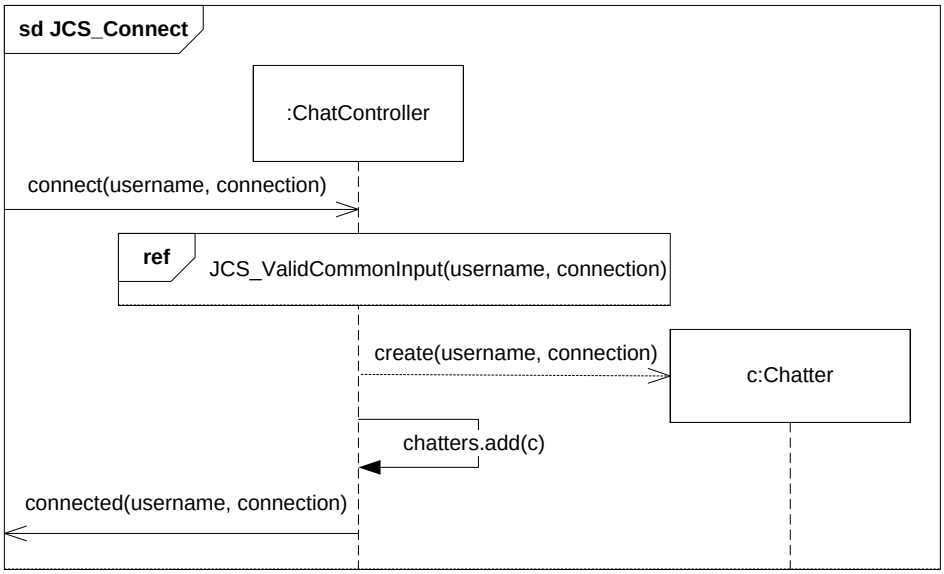
En tilsvarende måte å utrykke det samme på er vist i figur 5-2 hvor jeg eksplisitt har avgrenset området av diagrammet som skal overvåkes.



Figur 5-2 Eksempel på bruk av try

Før vi kan gå i gang med å spesifisere connectDefault, må vi vite hvilke unntak connectDefault skal kunne håndtere. Fra Java så er man vant med at metoder spesifiserer hvilke unntak de kan kaste. Så langt har jeg ikke innført en tilsvarende mekanisme i sekvensdiagrammer, men det kunne være aktuelt for å gjøre det enklere å forsikre seg om at man har håndtert alle relevante unntak. Dette er likevel noe som kan sjekkes uten egne mekanismer, og er derfor så langt utelatt.

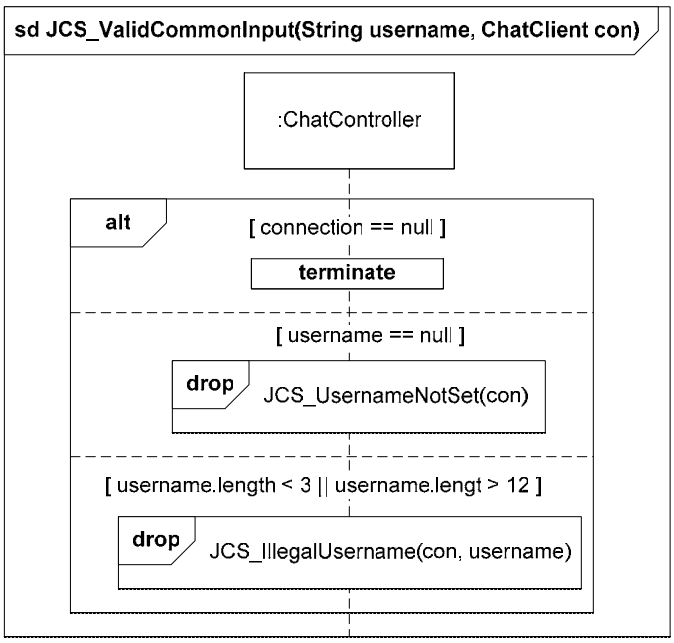
I JCS_Connect (se figur 5-3) har jeg lagt til en referanse til JCS_ValidCommonInput hvor det evalueres for eventuelle unntak.



Figur 5-3

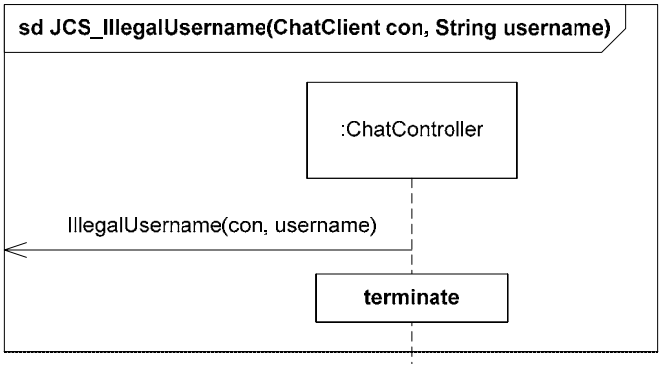
I JCS_ValidCommonInput (se figur 5-4) sjekker jeg på om noe har gått galt, og eventuelt slipper (drop) et unntak. Å slippe et unntak innebærer at man fortsetter tracen i et nytt diagram på et lavere nivå. Drop er i dette tilfellet en spesialisert referanse. Ved å gå til et

lavere nivå får man nye gater, og dermed kommer vi oss rundt problemet med gate matching. Vi så sende ut uventede meldinger til defaults.



Figur 5-4

Neste steg blir nå å utforme diagrammene som drop referer til. Figur 5-5 viser hvordan man sender en uventet melding for et ulovlig brukernavn. Jeg viser ikke i modellen hvem som skal motta unntaket. Jeg sier kun hvilken melding som sendes ut, og hva som skjer med livslinjen til avsender etter at meldingen er sendt. I dette tilfellet så termineres eventuell videre trace fra ChatController.



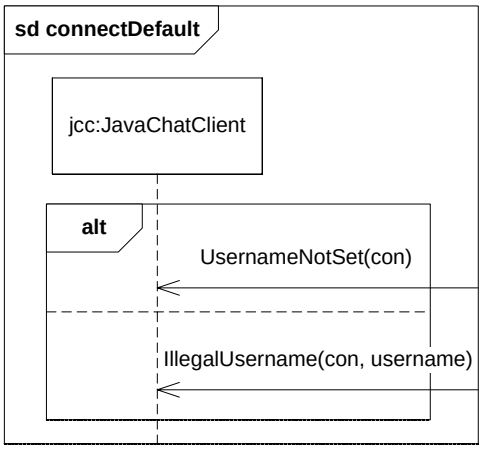
Figur 5-5 Eksempel på sending av uventet melding

Et stort spørsmål er nå hvordan Illegalusername kommer seg fra ChatController til en gitt JavaChatClient. For å kunne vite dette så må man se på hvilke defaults som omslutter den aktuelle delen av tracen. Som vist i figur 5-2 er det kun definert en default, connectDefault, som overvåker tracen. I denne har vi også en gate match for IllegalUsername meldingen. Hadde man hatt flere defaults måtte man ha sjekke alle for

eventuelt passende. En unntaksmelding finner altså rett default ut fra mengden av defaults som overvåker tracen, og hvilke gates disse har. Finnes det flere defaults med passende gate er det vilkårlig hvem som håndterer unntaket.

I forhold til gate matching på 1. etasjen (normalflyten), så får ikke default'en problemer med disse fordi en default i denne sammenhengen representerer en underetasje. Hver underetasje har sine egne gater. Fra en gitt livslinje kan man alltid se den nærmeste underetasjen i form av en default eller drop. Hver default kan, ved behov, ha sine egne underetasjer.

Når det gjelder hvordan Illegalusername meldingen blir håndtert er dette vist i figur 5-6.



Figur 5-6 connectDefault

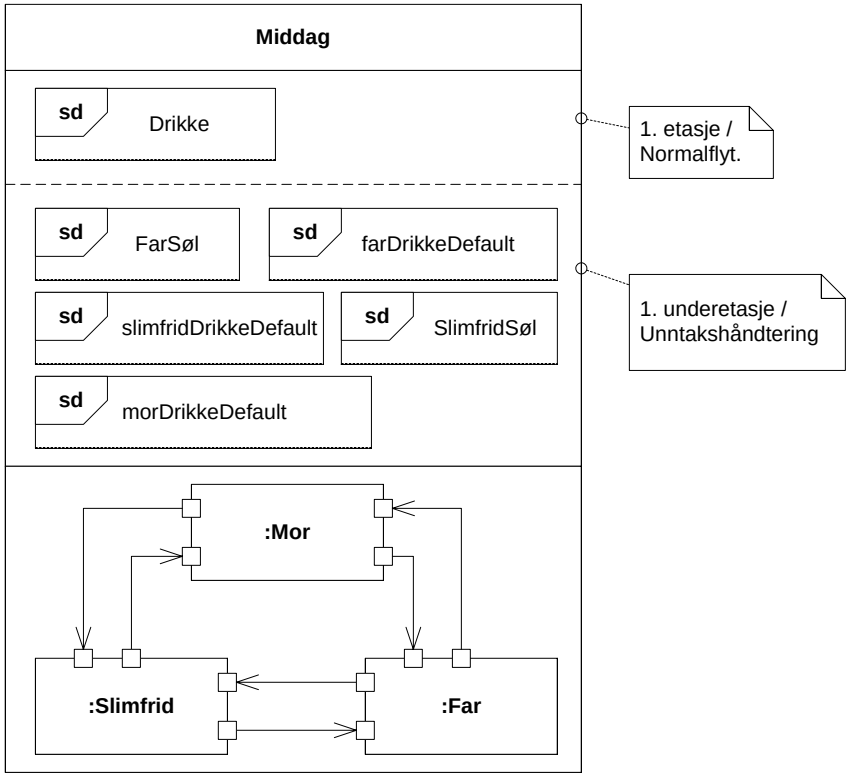
For å kunne sende meldinger til de gitte underetasjene må man bruke en drop. Om man bruker en drop mens man er i en underetasje går man ytterligere en etasje ned. Det er altså samsvar mellom antall drop man tar, og hvilket nivå aktuelle defaults må være på. Hadde jeg i figur 5-5 også hatt en drop, ville resultatet blitt at jeg gikk ned enda et nivå. Dette er aktuelt der hvor håndtering av unntak i seg selv kan resultere i unntak.

Resultatet er at man får en stram struktur i forhold til gater til tross for en lagdeling. Har man x antall drop på samme nivå, så representerer dette en alt struktur i forhold til hvordan mottaker må organisere sin default, som vist i figur 5-6.

5.2. Et mer komplisert eksempel

Nå som jeg har presentert mekanismene vil jeg her presentere et mer komplisert eksempel som trekker inn flere problemstillinger. Eksempelet her vil ta utgangspunkt i en familiemiddag hvor det er 3 aktører, mor, far og deres datter. Sekvensdiagrammet (se figur 5-8) er del av en middag, og i dette diagrammet skjenker mor opp drikke til far og datter. Aktuelle unntak som i dette tilfellet kan oppstå er at far eller datter registrer at det er sølt drikke.

I kapittel 5.1 presenterte jeg ikke noen form for kontekst, eller satte diagrammene for unntakshåndtering inn i en større sammenheng. I figur 5-7 viser jeg hvordan man kan få visualisert denne lagdelingen i en kompositt struktur, og på denne måten også få deklarert samtlige sekvensdiagrammer.

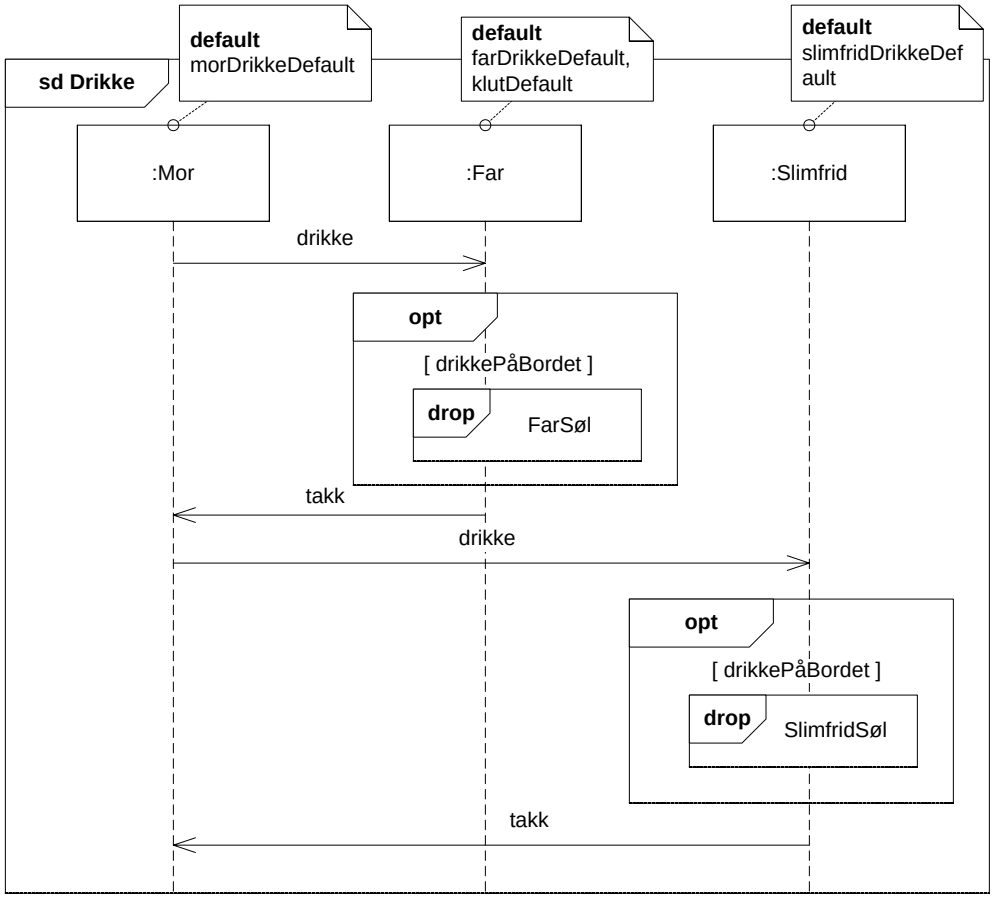


Figur 5-7 Konteksten til middagen

Denne strukturen har en liten utvidelse i forhold til hva som tidligere er vist. Jeg har her lagt til en lagdeling hvor de ulike nivåene med normalflyt og unntakshåndtering er lagt til. Ettersom det i dette eksempelet kun vil være én underetasje med unntakshåndtering, ble det her kun to etasjer. Det kan i utgangspunktet være så mange etasjer man selv måtte ønske.

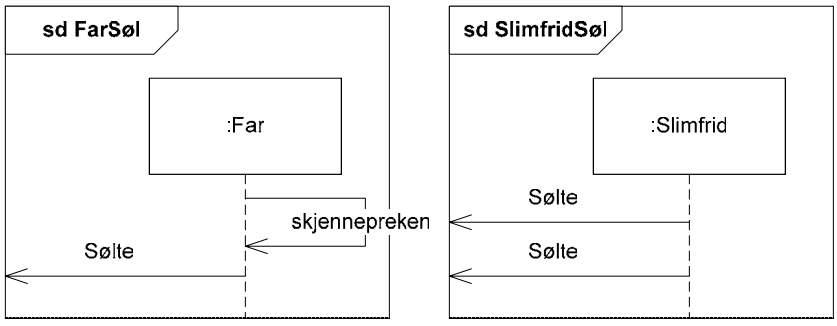
Fordelen ved å skille på etasjer allerede her er at man får et raskt overblikk over hva som hører til unntakshåndtering og hva som hører til normalflyten. En videre utvidelse kunne vært å også legge til en form for *Exception Overview Diagram*, noe lignende et *Interaction Overview Diagram*. Dette diagrammet ville trolig kunne hjelpe til med å holde oversikt over de ulike kontrollflytene for unntakshåndtering, men jeg har ikke tatt med noe sånt her i denne omgang.

Når det gjelder modellen i figur 5-8 byr den på flere utfordringer med tanke på unntakshåndtering. For det første så kan både far og datter sende melding om søl, avhengig av hvem som søler. Dernest så er også dette en typisk situasjon hvor gjerne flere fanger opp situasjonen. Sett at man sitter rundt middagsbordet, og datteren plutselig utbryter at det er sølt, da vil alle rundt bordet fange opp situasjonen.



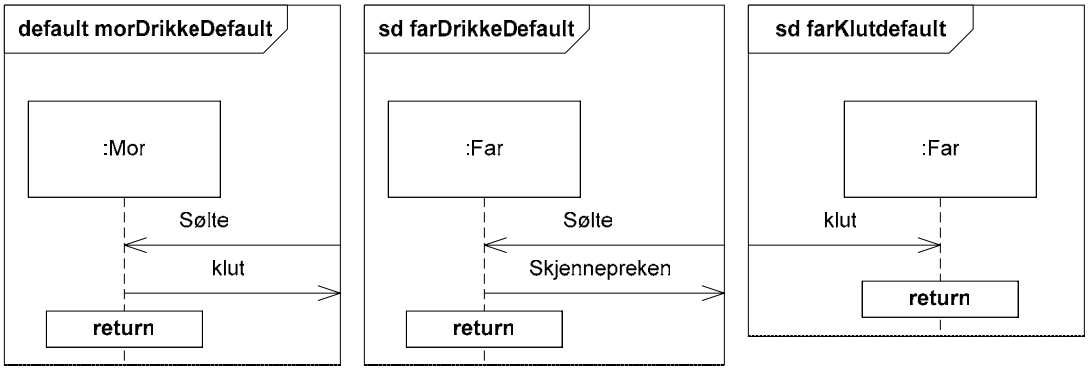
Figur 5-8 drikke eksempelet.

For det første så må man her være i stand til å håndtere at unntak kan oppfattes av flere livslinjer. Da sekvensdiagrammer ikke har mulighet til å sende multicast meldinger, må man eksplisitt sende ut korrekt antall meldinger selv. Dette innebærer at man må sende en melding til hver mottaker, i dette tilfellet maks 2 meldinger.



Figur 5-9 Eksempel på Søl diagram

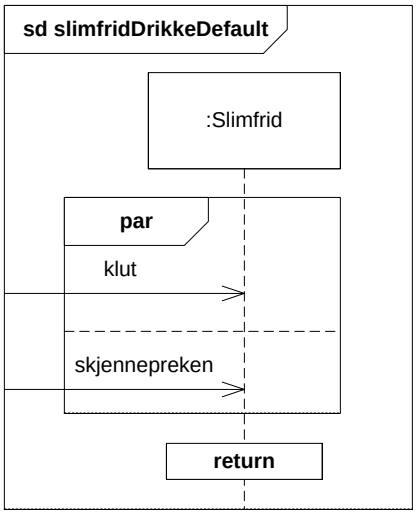
Når det gjelder mottak av meldingene om søl er det kun mor og far som kan ta imot meldinger om søl. Deres defaults er som følger.



Figur 5-10 Håndtering av søl

Jeg kunne fra FarSøl ha sendt ut 2 sølte meldinger, og latt farDrikkeDefault håndtere den ene, men ville da fått problem med *return*. Grunnen til det er at jeg da ville ha signalisert return i både farDrikkedeFault og defaulten som far ville brukt for å ta imot skjennepreken på. Dette ville resultere i at far hadde signalisert return før unntakshåndteringen var ferdig, noe som ville resultert i ulovlige tracer.

Når det gjelder datteren sin default må denne kunne ta imot både klut og skjennepreken samtidig, før normalflyten kan fortsette.



Figur 5-11 Datter sin håndtering av søl

Med denne måten å uttrykke unntakshåndtering på, blir det en utfordring å få et godt overblikk over kontrollflyten. Grunnen til dette er den dynamiske koblingen av gater som resulterer i et stort antall mulige tracer.

Dette gir store fordeler med tanke på løs kobling mellom det å sende og motta unntak, men som sagt er det problematisk å få et godt overblikk over de lovlige tracene. Grunnen til at det er ekstra problematisk å følge kontrollflyten ved asynkron unntakshåndtering er

at det ikke er noe reelle begrensninger på hvor unntak kan sendes, sett i forhold til Java exceptions hvor dette ligger pent lagret på kall stakken. Dette gjør at man i sekvensdiagrammer må være nøye med å sjekke grensene for overvåkingen, for å kunne følge tracen gjennom mengden av drop og defaults.

For å kunne lage en lovlig trace så må videre alle gater matche. Det innebærer at om far søler, så vil eneste lovlige trace være:

FarSøl seq MorDrikkedefault seq FarDrikkeDefault.

Ved denne tracen får man ingen gater til overs, og vi har dermed en komplett og lovlig trace. I kapittel 5.3 vil jeg beskrive nærmere hvordan man kommer fram til lovlige tracer.

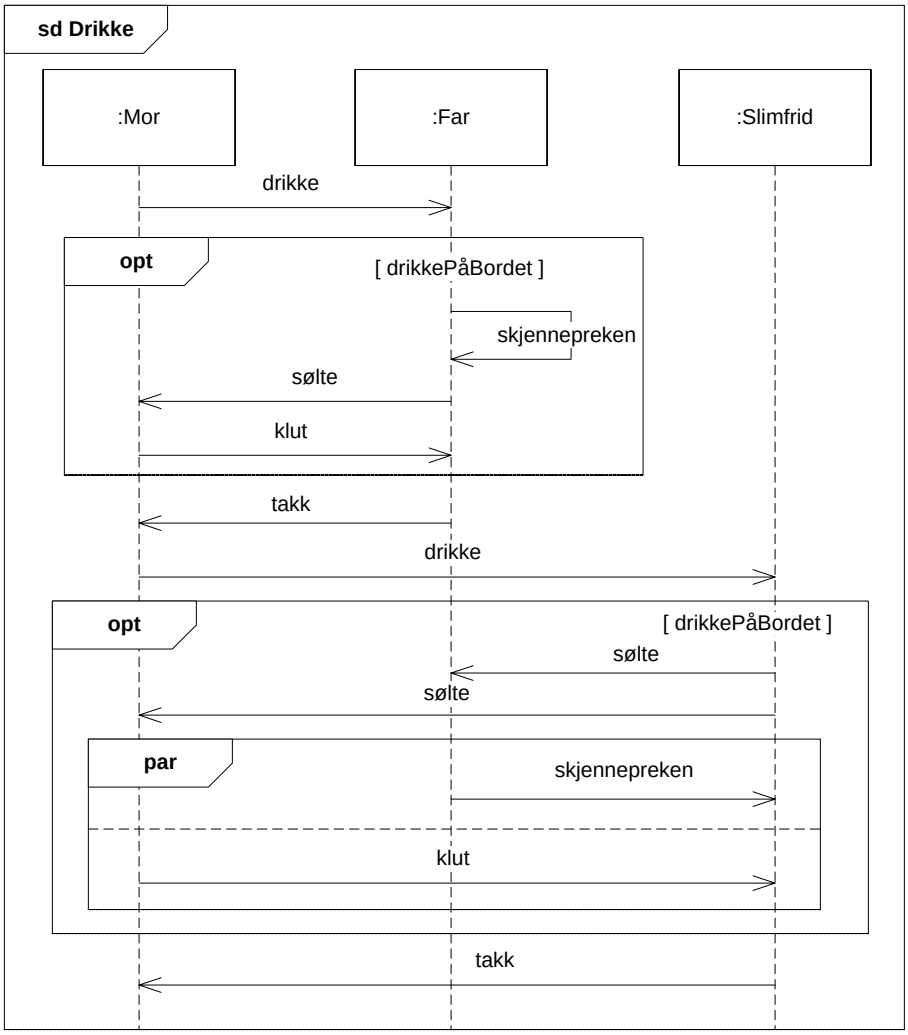
5.3. Presis semantikk for mekanismene

Jeg har nå vist hvordan man kan utrykke unntakshåndtering i sekvensdiagrammer, og gitt to eksempler på bruken av mekanismene. Jeg vil nå gå videre med å se mer i detalj på hva mekanismene har å si for de eksplisitte tracene til sekvensdiagrammene. Jeg kommer her til å bygge videre på trace semantikken Haugen et al. (2003) presenterte i STAIRS metodologien. Dette er en noe mer formalisert utgave av semantikken presentert i UML 2.0 spesifikasjonen (2004).

For det første så må sekvensdiagrammene støtte svak sekvensialisering. Veldig kort så betyr det at rekkefølgende på hendelser må være på en sånn måte at man sender en melding før den kan mottaes, og hendelser på hver livslinje må være ordnet i den rekkefølgen de står oppgitt. I forhold til figur 5-8 betyr det at *Mor* må sende *drikke* før *Mor* kan motta *takk* fra *far*, og at *Far* må motta drikke før *takk* kan sendes.

Videre så kommer jeg til å benytte meg av tilsvarende notasjon som Haguen et al. (2003) brukte, ved å si at sending av melding angies som *!mldNavn*, og mottak av melding angies som *?mldNavn*.

Som eksempel vil jeg fortsette å bruke middags eksempelet fra kapittel 5.2. For å kunne ha noe å sammenligne tracene fra modellen med unntakshåndtering, vil jeg her presentere samme modellen uten bruk av unntakshåndtering (se figur 5-12). Grunnen til at jeg ønsker å sammenligne tracene er for å se om de utrykker det samme, og eventuelt hva som blir forskjellig.



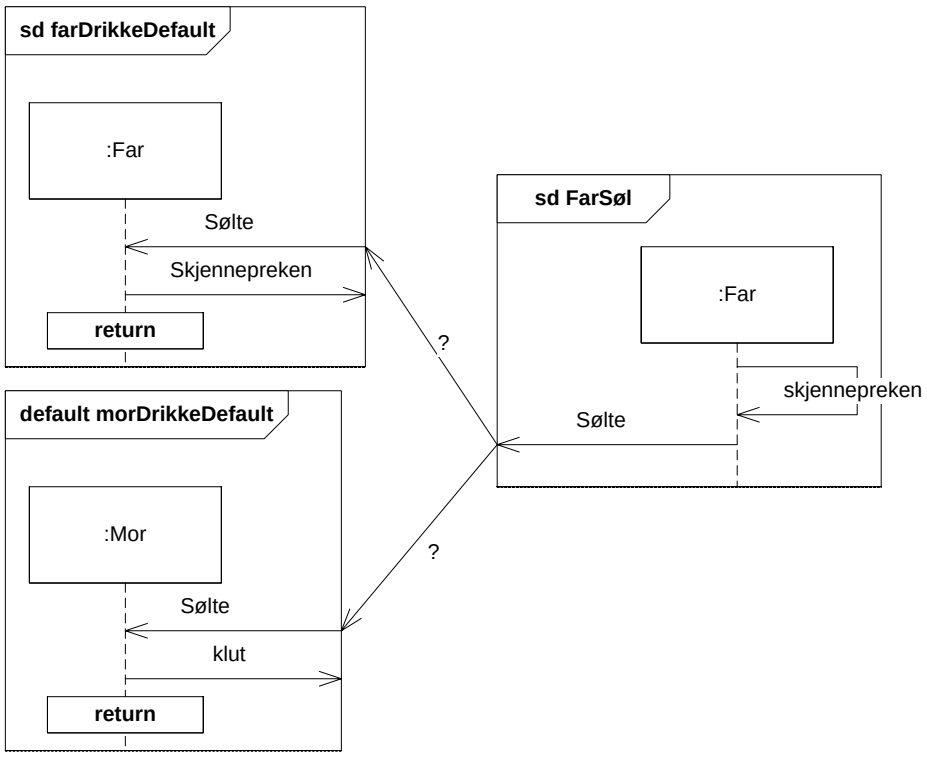
Figur 5-12 Drikke eksempelet uten unntakshåndtering

Modellen representerer følgende trace:
Drikke = <!drikke, ?drikke> seq
opt (<!skjennepreken, ?skjennepreken, !sølt, ?sølt, !klut, ?klut>) seq
<!takk, ?takk, !drikke, ?drikke> seq
opt((<!sølte, ?sølte> seq <!sølte, ?sølte>) seq
(<!skjennepreken, ?skjennepreken> par <!klut, ?klut>)) seq
<!takk, ?takk>

Denne tracen bør vi også være i stand til å få fram fra modellen for unntakshåndtering. Hvis vi gjør det, vet vi at modellene kan utrykke det samme. Det som kan tenkes er at mekanismene for unntakshåndtering også utrykker andre tracer i tillegg, men det kommer jeg tilbake til.

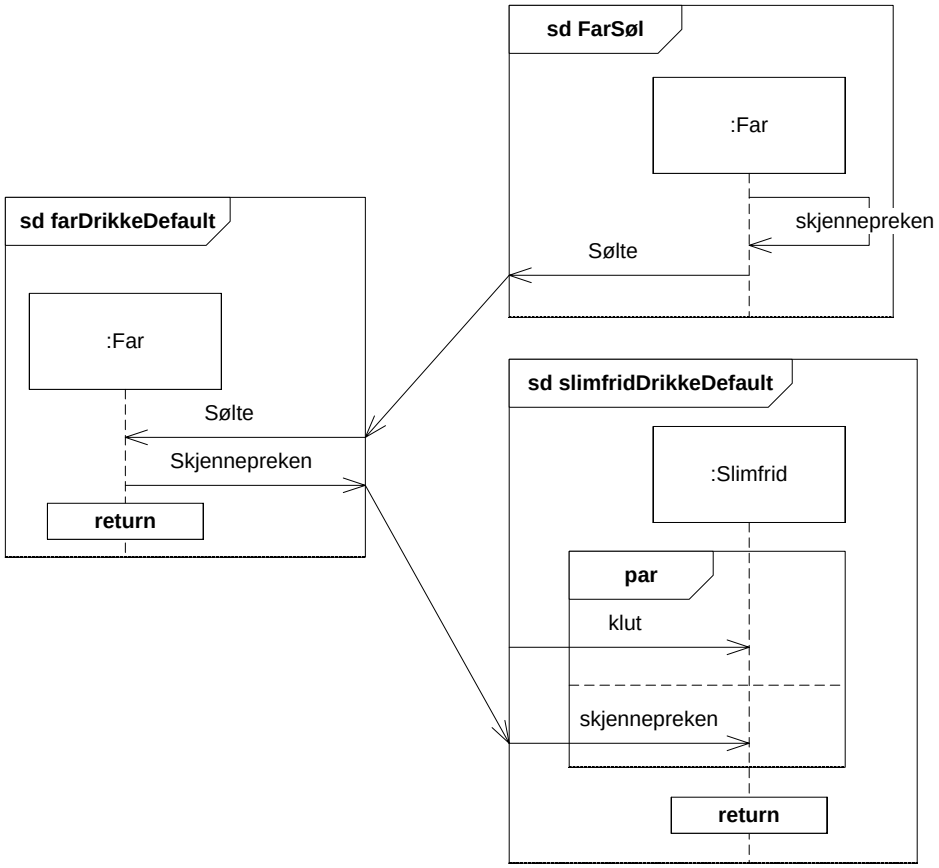
Før jeg går i gang med å utlede mulige tracer for unntakshåndteringen vil jeg gå tilbake og beskrive nærmere hvordan man finner fram til lovlige tracer for unntakshåndtering. Hvis vi nå ser for oss at far søler. Da starter unntakshåndteringen i diagrammer sd FarSøl.

I dette tilfellet så sender far ut meldingen *sølte*. Neste spørsmål da er hvilken default som fanger opp denne meldingen.



Figur 5-13 sd FarSøl

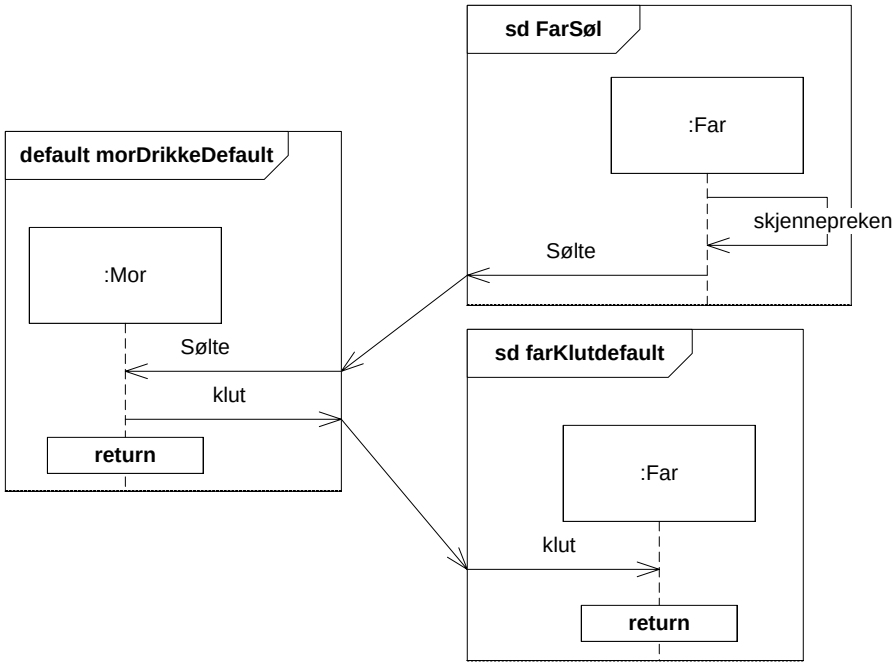
Ser man på figur 5-13 oppdager man at det er 2 mulige mottakere, både Far selv og Mor. Jeg sa tidligere at for å få en lovlig trace så må alle gater match. For å finne den som er lovlig må vi følge gatene fra farDrikkeDefault og morDrikkeDefault videre. Om vi først følger meldingen *Skjennepreken* oppdager vi at det kun er datteren som kan ta imot denne meldingen.



Figur 5-14 Ugyldig gate match

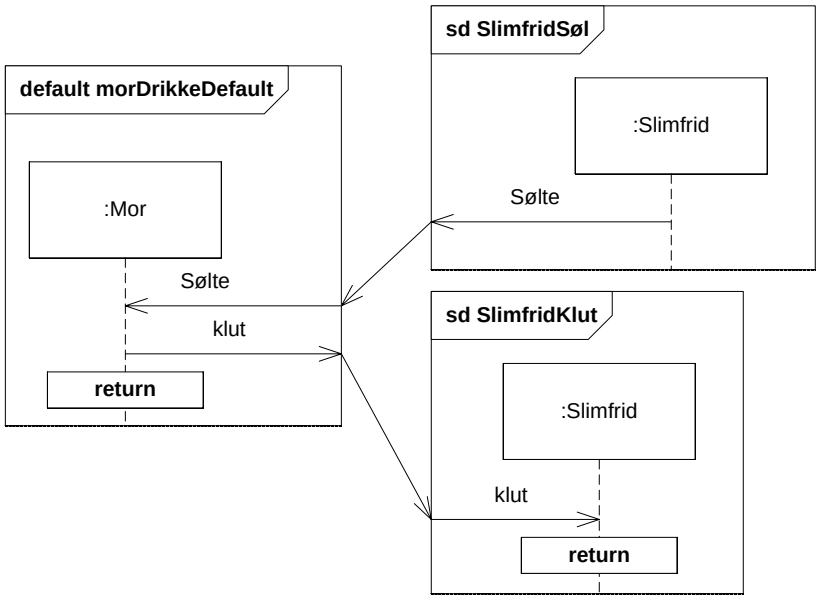
Slimfrid kan ta imot skjennepreken, men hun skal også kunne ta imot klut samtidig, og klut mottar hun ikke fra noe sted. Dermed har vi fått en gate til overs, og tracen er ulovlig. Det betyr at den andre mulige tracen må være den gyldige, forutsatt at det finnes en gyldig trace for unntakshåndteringen.

Den andre mulige tracen er vist i figur 5-15. Hvis man sjekker gates'ene der finner man at det ikke er noen gates til overs, og vi har dermed en lovlig trace for unntakshåndteringen nemlig: FarSøl seq morDrikkedefault seq farKlutdefault



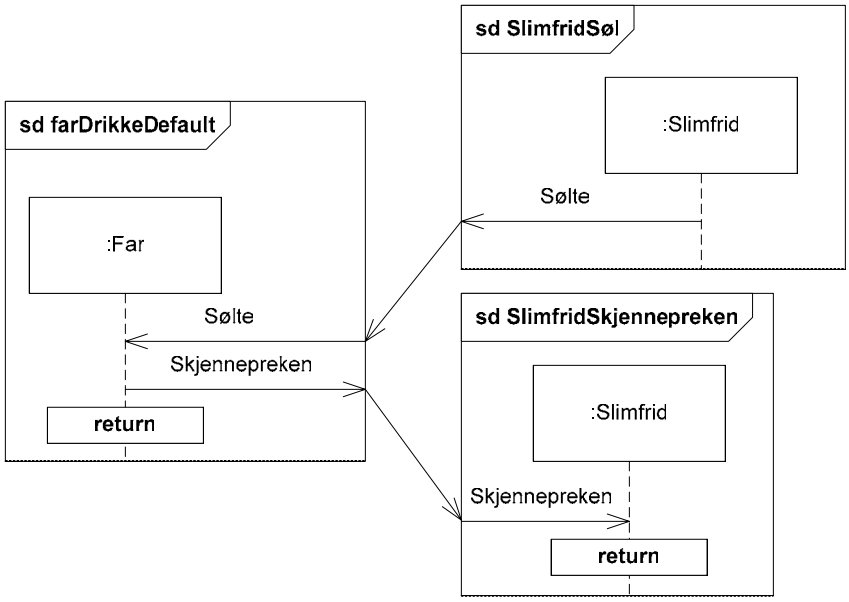
Figur 5-15 Gyldig trace for unntak

Et annet tilfelle er om man har flere lovlige tracer å velge blant for en unntaksmelding. Det kunne tenkes at Slimfrid kun sendte ut én melding. Da ville vi fått 2 potensielle tracer (se figur 5-16 og figur 5-17), forutsatt at Slimfrid hadde de nødvendige defaultene.



Figur 5-16 Potensiell trace

I figur 5-16 og figur 5-17 har jeg for eksempelet sin skyld nå lagt til to nye defaults SlimfridKlut og SlimfridSkjennepreken.



Figur 5-17 Annen potensiell trace

I dette tilfellet så er det ingen regler som angir om unntaket når Mor eller Far. Det er altså vilkårlig om Slimfrid får en klut eller skjennepreken om det blir sølt. Dette er kun mulig å få til ved en dynamisk kobling av gater, og ville vanskelig latt seg gjennomføre uten mekanismene for unntakshåndtering.

Tilbake til hovedeksempelet (se figur 5-8) så sa jeg at en gyldig trace for unntaket at far oppdager søl er:
drop FarSøl = FarSøl seq morDrikkedefault seq farKlutdefault

Mer detaljert gir tracen følgende:
drop FarSøl = <!skjennepreken, ?skjennepreken, !sølt, ?sølt, !klut, ?klut>

For tilfellet at Slimfrid oppdager søl blir tracen:
drop slimfridSøl = SlimfridSøl seq morDrikkeDefault seq farDrikkeDefault
seq slimfridDrikkeDefault

Mer detaljer gir tracen følgende:
drop slimfridSøl = (<!sølte, ?sølte> seq <!sølte, ?sølte>)
seq
(<!skjennepreken, ?skjennepreken> par <!klut, ?klut>)

Tracen for sekvensdiagrammet i figur 5-8 blir da som følger:

Drikke = <!drikke, ?drikke> seq
opt (drop FarSøl) seq
<!takk, ?takk, !drikke, ?drikke> seq
opt(drop SlimfridSøl) seq
<!takk, ?takk>

Tar man nå og bytter ut *drop FarSøl* og *drop SlimfridSøl* ser man at man har utrykk akkurat det samme som vist uten mekanismer for unntakshåndtering. Jeg utleder ikke her tracene i detalj. For å gjøre det må man blant annet gå tilbake til hva jeg beskrev om semantikken for drop i kapittel 4.4.2. Essense av det var at *drop* la til en *opt* i forhold til resten av tracen, hvor denne *opt'en* ble utført om man signaliserte *return* og hoppet over hvis man signaliserte *terminate*.

Jeg har nå vist at mekanismene for unntakshåndtering gir de samme tracene som modellene uten. Dermed kan jeg slå fast at jeg har klart å skape et visuelt skille og løs kobling mellom sending og mottak av unntak, uten at tracen har endret seg.

5.4. Utvidelse av trace semantikken

Konklusjonen jeg gav i 5.3 tok en forutsetning, som jeg ikke nevnte, og det var at deler av normalflyten ikke kunne fortsette i parallell med unntakshåndteringen. Meldingsutveksling mellom livslinjer som ikke er påvirket av unntakshåndteringen vil i praksis blande seg inn med unntakshåndteringen, og dette er en av styrkene til mekanismene. Dette legger til rette for økt parallellitet i unntakstilfeller, som standard UML 2 sekvensdiagrammer ikke klarer å utrykke like enkelt. I forhold til hvordan man utleder disse tracene så har jeg dessverre ikke fått sett ordentlig på dette, og ettersom sekvensdiagrammer kun viser deler av en stor helhet anser jeg ikke dette som noe stort problem.

Noen løse tanker rundt denne problemstillingen har jeg likevel gjort meg. For å kunne utlede komplette tracer for denne parallelliteten er det helt sentralt at man har oversikt over hvilke livslinjer som er påvirket av unntakshåndteringen og hvilke som ikke er det. Alle livslinjer som er del av sekvensdiagrammet og som ikke er påvirket av unntakshåndteringen kan intuitivt fortsette som om ingenting er skjedd, helt til de selv blir påvirket. De som er påvirket kan derimot ikke fortsette normalflyten, før de eventuelt blir upåvirket.

Med påvirket så mener jeg følgende:

- Har utført en drop
- Er i en default, eller har utført en default

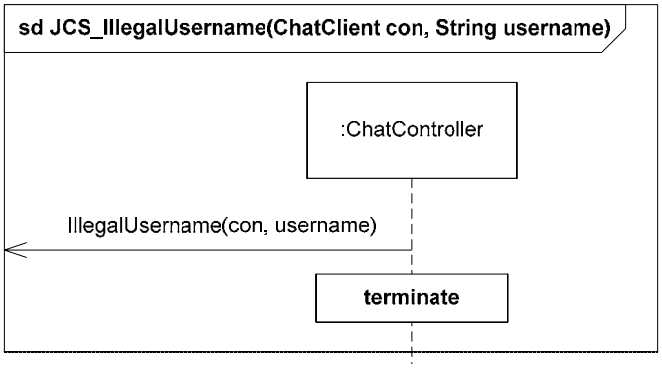
Eneste måten å komme seg ut av en påvirket tilstand er ved å signalisere return. Signaliserer man return går livslinjen man signaliserte return på, tilbake til upåvirket tilstand og kan fortsette normalflyten. Dette innebærer at man alltid må være varsom med bruk av return, og passe på at all potensiell unntakshåndtering på denne livslinjen er utført før man kaller return.

Det positive er at dette medfører en større grad av parallellitet enn hva man ville klare å få til uten bruk av mekanismer for unntakshåndtering. På den andre siden så medfører dette veldig kompliserte tracer, og algoritmen for å komme fram til denne mengden av tracer har jeg som nevnt ikke fått utformet. Tracene jeg presenterte tidligere forutsatte at det ikke var noen som helst form for parallellitet mellom unntakshåndtering og

normalflyt, dette var heller ikke tilfellet i eksempelet, og begrunnet som nevnt dette med at sekvensdiagrammer kun viser en del av den store helheten. Ettersom sekvensdiagrammer ikke viser alt kan man derfor velge å fokusere på det som er mest sentralt.

5.5. Forenklinger

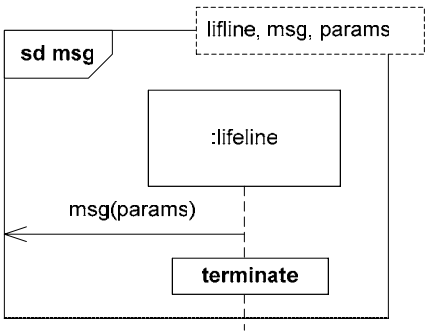
I mange tilfeller så vil en drop gjerne referere til et diagram som ser ut som noe tilsvarende det som er vist i figur 5-18.



Figur 5-18 Eksempel på et typisk diagram drop referer til

Man ønsker altså kun å sende ut et unntak og så terminere. I lengden så er det tidkrevende, og ganske unødvendig å modellere dette for alle tilsvarende unntak. Dette tvinger fram en forenkling. I noen tilfeller så ønsker man kanskje også å utrykke at forskjellige livslinjer kan bruke samme drop. Sånn som i middagseksempelet, så ønsket både far og datter å kunne si ifra om at det var sølt. I det tilfellet så lagde jeg 2 ulike diagrammer, FarSølte og SlimfridSølte. Man kunne tenkt seg at far og datter i dette tilfellet ønsket å gjøre det samme og derfor kunne man tenkt seg at de begge to ønsker å slippe søl.

I begge disse tilfellene så må man være i stand til å tilpasse sekvensdiagrammet. Både når det gjelder navn, melding, parametre og livslinje. Til dette kan vi for eksempel benytte oss av UML 2 Templates, som vist i figur 5-19



Figur 5-19 Eksempel på et template for å sende unntak

Detaljene rundt hvordan man eventuelt sømløst kobler sammen drop og templates har jeg ikke fått sett nok på. Det som er klart er at det er høyst aktuelt å benytte seg av templates for å redusere behovet for å lage diagrammer for å sende unntak. Grunnen til at man bør koble inn bruk av templates her, framfor å si at sending av unntak kan skje automatisk er at man risikerer for stor sammenblanding operatorer. En drop har mer til felles med en ref enn å sende en melding. Hvis drop i noen tilfeller automatisk skal kunne send unntak, er det fare for at man legger for mye funksjonalitet i drop konstruksjonen. Med dette utgangspunkt er det å trekke inn templates for å komme rundt utfordringen en bedre løsning.

5.6. Oppsummering

Jeg har i dette kapittelet presentert mitt forslag til mekanismer for unntakshåndtering i sekvensdiagrammer. Forslaget baserer seg på en blanding av planløsningen jeg tidligere skisserte og U2TP sin default mekanisme. Mekanismene løste flere utfordringer med tanke på at mekanismene skulle være kompakte og gi en løs kobling mellom sending og mottak av unntak og ikke minst skape et visuelt skille. Videre gir de mulighet for å overvåke kjøringer i tillegg til å knyttes oss mot bestemte hendelser.

Videre så påpekte jeg at det er utfordringer med tanke på å holde oversikt over kontrollflyten, og at dette er noe som bør jobbes videre med. Jeg presenterte også en presis semantikk, og tok fram ulike problemstillinger knyttet til dette. Deretter så jeg på mulige forenklinger i forhold til bruken av mekanismene, spesielt med tanke på bruk av templates for å redusere antall diagrammer for unntakshåndtering.

Selv om kontrollflyten er noe komplisert, vil jeg konkludere med at mekanismene for unntakshåndtering har møtt de målene og kravene som jeg satte meg.

6. Metodikk for avdekking av unntak i sekvensdiagrammer

Jeg vil her ta for meg siste del av problemstillingen, som går på å utforme et forslag til metodikk for bruk av unntakshåndtering i modellering. Jeg vil beskrive hvordan man bør gå fram for å avdekke unntak og hvordan man kan strukturere mengden av unntak. Til slutt vil jeg presentere et eksempel på bruk av metodikken. Da jeg så langt har hatt fokus på sekvensdiagrammer vil jeg også bruke sekvensdiagrammer som utgangspunkt for denne metodikken.

6.1. Finne mulige unntak

For å avdekke mulige unntak vil jeg ta utgangspunkt i en teknikk fra sikkerhetsanalyse, nemlig *Hazard and Operability analysis* (HazOp)[3]. HazOp har blant annet har som mål å avdekke avvik fra ønsket oppførsel, og det er akkurat hva vi her ønsker. Jeg vil gjøre noen endringer i HazOp analysen for å få den til å passe bedre overens med unntakshåndtering, uten å gå inn på hva jeg endrer.

For å kunne bruke denne metodikken kreves det at man modellerer i 2 omganger, men med ulike fokus. Første runden med modellering går på å spesifisere normalflyten. Eksempel på spesifisering av normalflyt finner du i kapittel 2, hvor jeg har spesifisert normalflyten for JavaChat systemet.

Andre runden går på å utvide denne normalflyten med mekanismer for unntakshåndtering. Det er det å utvide normalflyten med mekanismer for unntakshåndtering jeg nå vil ta for meg i denne metodikken.

For å avdekke unntak følger man følgende generelle framgangsmåte:

1. Velge ett sekvensdiagram
2. Gå systematisk igjennom diagrammet melding for melding
    a. For hver melding anvend en gitt mengde ledeord med tanke på å avdekke eventuelle unntak, for eksempel:
        i. Ulovlig
        ii. Null verdi
        iii. Utilgjengelig
        iv. Ukjent
        v. Eksepsjonelt
        vi. Eksisterer
3. Lag en unntakstabell for hver livslinje (tabell 6-1). For hver påpekte unntak, list dette opp tabellen med navn på melding som kan resultere i ett unntak, betingelsen for unntaket, hvilket diagram unntaket skal slippes til og om det skal flyten skal termineres eller fortsettes og kort kommentar om hvordan unntaket skal håndteres.

[3] Se Redmill et al. (1999) for detaljer om HazOp

4. Strukturer og grupper unntakene i et hierarki som gir mulighet for å legge til nye unntak på en sømløs måte.
5. Før inn funnene i modellene.

For å få dette klarer, vil jeg nå gå gjennom et eksempel på bruk av denne metodikken, og beskrive hvert punkt nærmere.

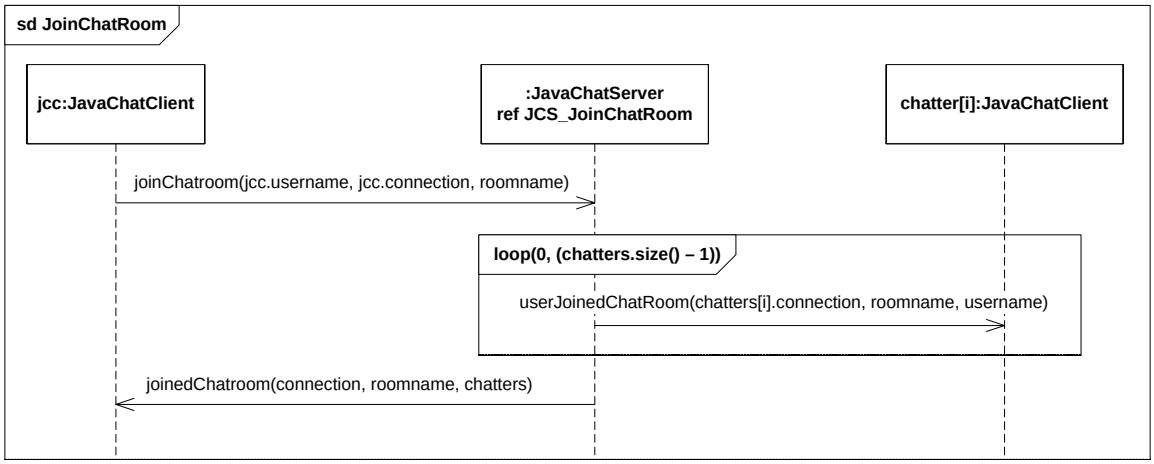
6.2. Eksempel på bruk av metodikken

Jeg vil her ta fram et par modeller fra JavaChat systemet og vise hvordan man kan gå fram for å avdekke eventuelle unntak i disse. Jeg kommer ikke til å vise alle mulige unntak, men unntak for noen av livslinjene.

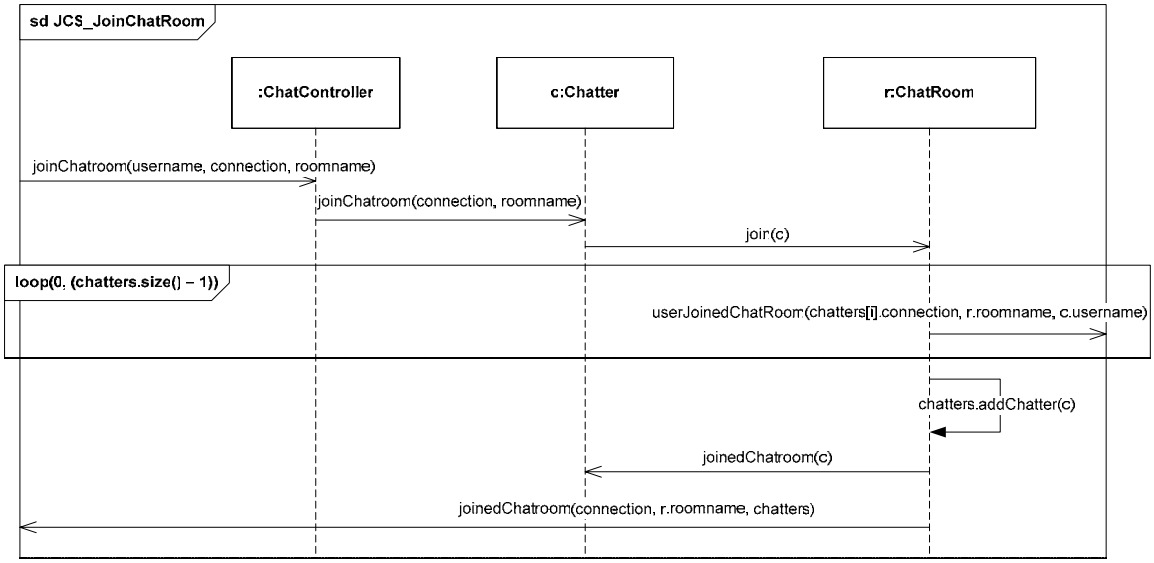
6.2.1. Velge sekvensdiagram

For å vise hvordan man avdekker eventuelle unntak i sekvensdiagrammer tar jeg for meg sekvensdiagrammene for å slutte seg til et praterom. Jeg går ut fra at modellene i utgangspunktet kun inneholder normalflyten. Det vil si at diagrammene inneholder det man ønsker at skal skje, og med gyldige parameter verdier.

Det ville ikke vært noe problem om diagrammene inneholder feilhåndtering, men da ville jeg anbefalt å fjerne dette før man la til unntakshåndtering for å gjøre modellene mer lett forståelige.



Figur 6-1 sd JoinChatRoom



Figur 6-2 JCS_JoinChatRoom

Det som figur 6-1 sd joinchatroom og figur 6-2 jcs_joinchatroom sier er at det normale er at en bruker får sluttet seg til et eksisterende praterom, og at brukere i dette rommet blir informert om den nye brukeren.

Jeg sa tidligere at man skulle velge ett sekvensdiagram som man skulle håndtere unntak i. Jeg har her presentert 2 sekvensdiagrammer, JoinChatRoom og JCS_ JoinChatRoom. Grunnen til dette er for å vise hvordan man skal velge sekvensdiagrammer. Hvis man hadde begynt med å spesifisere unntak for JoinChatRoom ville man ikke kommet så langt. Grunnen til dette er at eventuelle unntak som kan fanges i JoinChatRoom er et resultat av unntak som blir oppdaget og sluppet i JCS_JoinChatRoom. Vi er først nødt til å spesifisere hvilke unntak som kan bli sendt fra JCS_JoinChatRoom før vi kan håndtere disse unntakene i JoinChatRoom.

Man bør altså begynne å spesifisere unntak på det dypeste nivået, for så å jobbe seg utover, når man har flere nivåer med referanser.

6.2.2. Systematisk gjennomgang

Når det gjelder gjennomgangen av sekvensdiagrammene er det viktig å først sette seg opp en liste med ledeord. Disse ledeordene vil hjelpe deg til å avdekke typiske unntak, og på den måte sikre at du avdekker flest mulige. I dette eksempelet benytter jeg meg av ledeordene gitt i kapittel 6.1.

For å holde orden på alle unntakene som underveis blir avdekket vil jeg anbefale at man benytter seg av tabellen som er vist i tabell 6-1. Tabellen viser aktuelle unntak for JCS_JoinChatRoom.

Sequence diagram: JCS_JoinChatroom			
Lifeline: ChatController			
Event	**Condition**	**Action**	**Comment**
joinChatroom(username, connection, roomname)	Connection == null	Terminate	Avbryt tracen når ingen connection er sendt med.
joinChatroom(username, connection, roomname)	Username == null	Drop JCS_UsernameNotSet(connection), terminate	Informer om at brukernavn ikke var satt, og terminer.
joinChatroom(username, connection, roomname)	Roomname == null	Drop JCS_RoomnameNotSet(username, connection), terminate	Informer om at navnet på praterommet ikke var sendt med.
joinChatroom(username, connection, roomname)	Username.length < 3 || username.length > 12	Drop JCS_IllegalUsername(username, connection), terminate	Informer om at brukernavn må være mellom 3 og 12 tegn.
joinChatroom(username, connection, roomname)	!chatters.exists(username)	Drop JCS_UnknownChatter(username, connection), terminate	Informer om at brukernavnet er ikke er i bruk, eventuelt opprett en ny bruker.
joinChatroom(username, connection, roomname)	!chatrooms.exists(roomname)	Drop JCS_UnknownChatroom(roomname), return	Opprett praterommet.

Tabell 6-1 Unntakstabell for ChatController i sd JCS_JoinChatroom

Man må lage en tilsvarende tabell for hver livslinje som kan oppdage og sende unntak i et sekvensdiagram. Jeg har bygd opp tabellen rundt en hendelse (eng.: *event*) som angir hva som kan trigge unntaket. Deretter en betingelse (eng.: *condition*) som må tilfredsstilles for at et unntak skal kunne slå til. Deretter har jeg angitt en handling (eng.: *action*). Dette beskriver ikke så mye hva som skal gjøres, men angir hvor unntaket skal slippes, og om tracen skal termineres eller fortsettes. Kommentarfeltet (eng: *comment*) angir hva som skal gjøres når et unntak inntreffer.

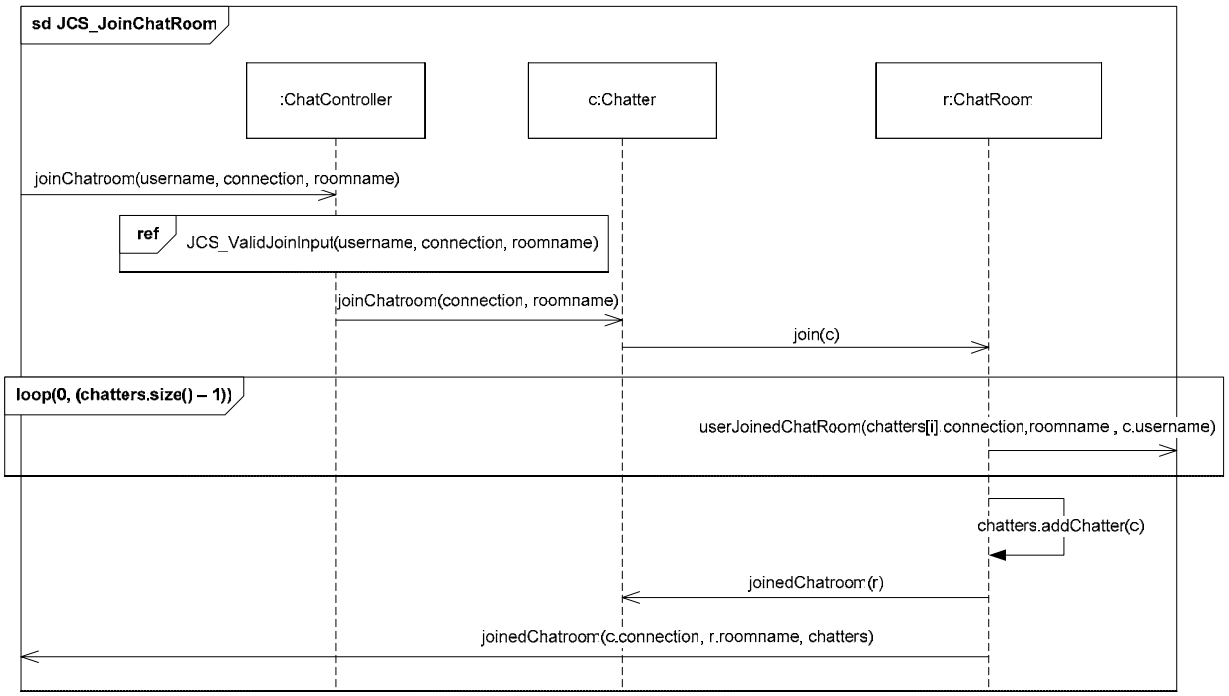
Jeg utelater her tabellene for Chatter og ChatRoom, men har i tabell 6-1 listet opp relevante unntak for ChatController. Her er det alt fra verdier som ikke er satt, verdier som allerede er tatt og noen som prøver å missbruke tjenesten. Det som er felles for alle disse er likevel at de evaluerer på en betingelse for å se om et unntak er inntruffet.

Tar vi nå for oss JoinChatRoom igjen, er vi ikke her interessert i å sette opp en tilsvarende tabell. For JoinChatRoom er vi kun interessert i å se på hvilke unntak JavaChatServer kan sende. Ut fra denne listen kan vi så designe oss en default for å ta

imot unntak. Hvordan denne default'en blir seende ut kommer jeg tilbake til i kapittel 6.2.4.

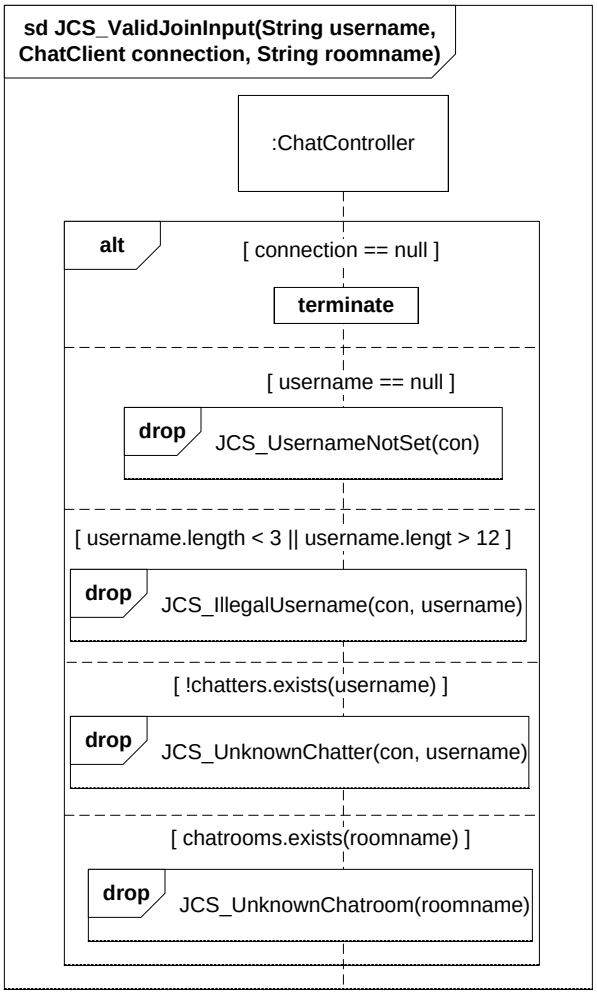
6.2.3. Legge til unntak i sekvensdiagrammet

Neste steg blir nå å legge unntakshåndtering til sekvensdiagrammene. Også her begynner vi på det dypeste nivået.



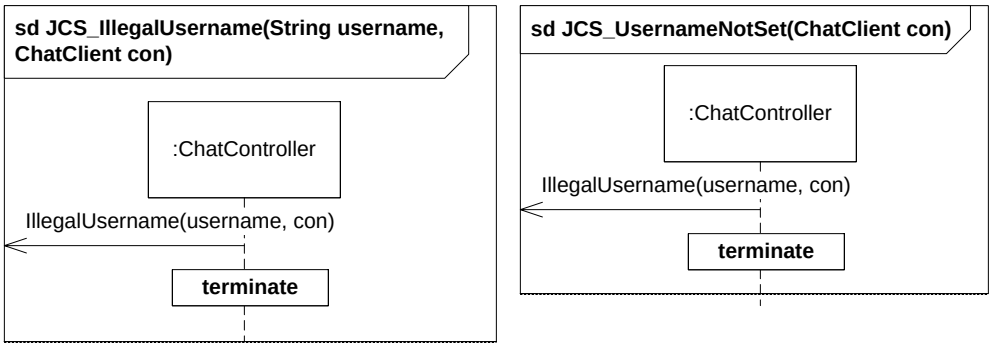
Figur 6-3 sd JCS_Join med ref'er til evaluering av unntak.

Som man ser er diagrammet fremdeles i realiteten som det opprinnelige bortsett fra at jeg har lagt til én referanser til henholdsvis JCS_ValidJoinInput.



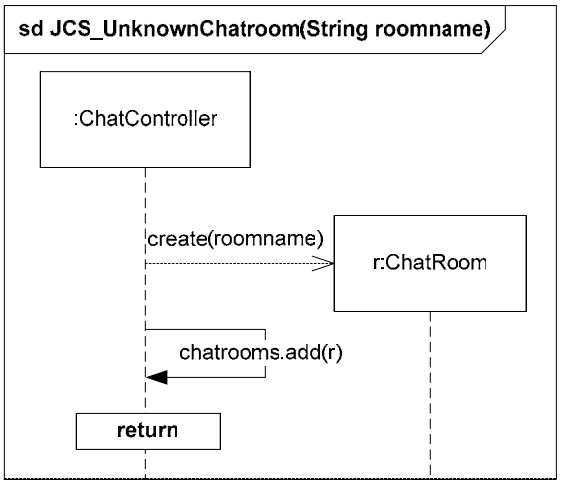
Figur 6-4 JCS_ValidJoinInput

Hvis en connection ikke er sendt med kan man ikke kommunisere med klienten, og da kan man må man bare avbryte tracen. Derfor slippes det der ikke noe unntak. Hvis brukernavnet ikke er satt, eller det er ulovlig så slippes unntakene som vist i figur 6-5.



Figur 6-5

Da unntaket med ukjent bruker ikke viser noe mer enn tilsvarende det som er vist i figur 6-5 utelates dette diagrammet fra dette eksempelet. Hvis noen derimot prøver å slutte seg til et ukjent praterom resulterer det i at praterommet blir opprettet som vist i figur 6-6.

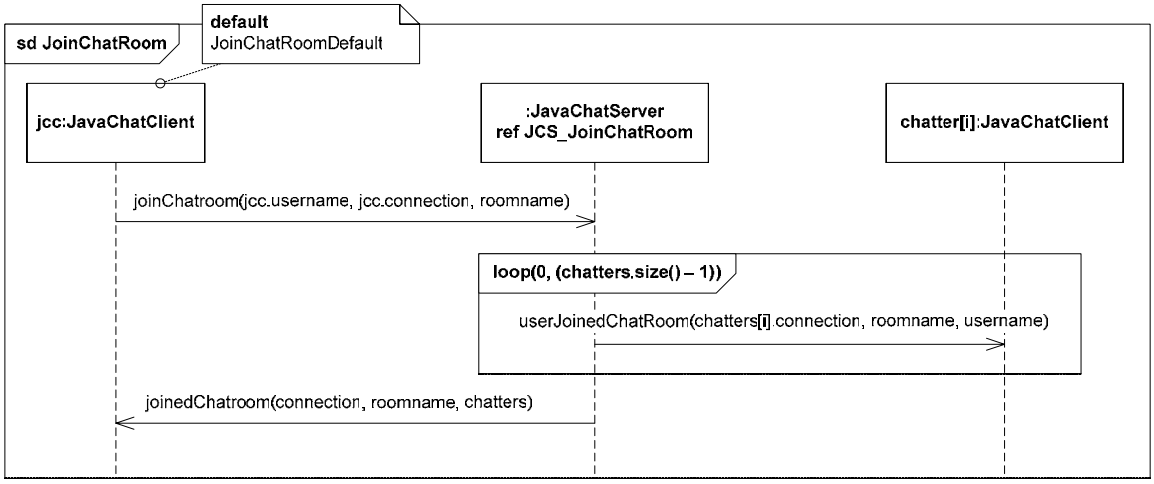


Figur 6-6 JCS_UnknownChatroom

Det ville ikke være noe i veien for at det som er vist i figur 6-6 godt kunne bli lagt inn som en *opt* i JCS_JoinChatroom. Grunnen til at jeg la det til som et unntak var at jeg ønsket å utrykke at man stort sett slutter seg til eksisterende rom. Derfor var det naturlig å gjøre oppretting av praterom til ett unntak.

6.2.4. Håndtere unntakene

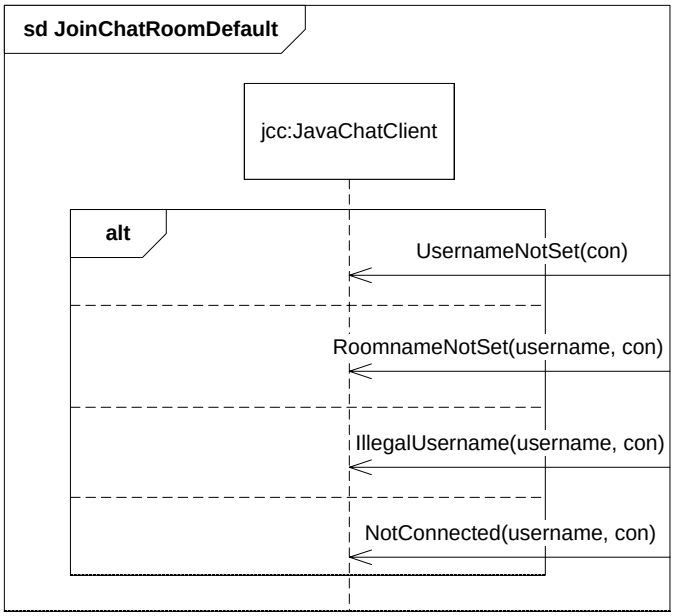
Det å håndtere unntakene innebærer at man legger til et plan for unntakshåndteringen. Ekstra plan legges til som defaults.



Figur 6-7 sd JoinChatRoom

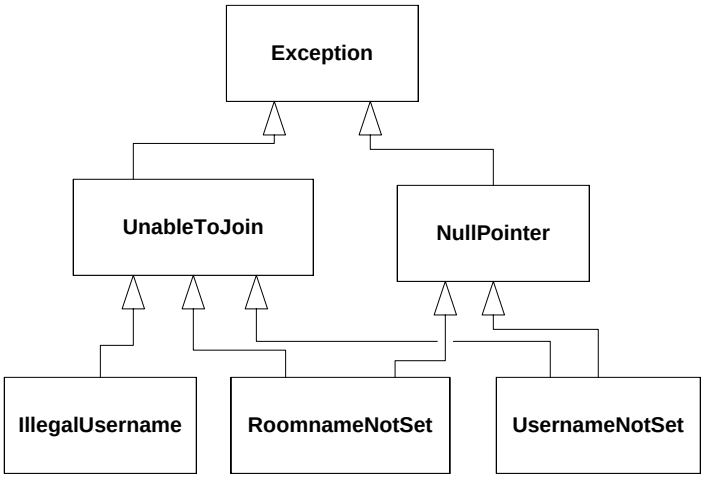
I figur 6-7 har jeg lagt til en default for å håndtere eventuelle unntak for det å slutte seg til et praterom. Det at jeg har koblet default'en direkte opp mot et objekt betyr at defaulten gjelder for hele livslinjen jcc:JavaChatClient i sekvensdiagrammet JoinChatRoom.

Selve defaulten JoinChatRoomDefault vil bli seende ut som vist i figur 6-8, i henhold til unntakene jeg tidligere spesifiserte. Flere unntak vil kunne komme til når man utvider med unntakshåndtering for de resterende livslinjene i JCS_JoinChatroom. Da jeg ikke har modellert JavaChatClient nøyer jeg meg her med å vise hvordan unntakene mottas og organiseres.



Figur 6-8 sd JoinChatRoomDefault

Noe jeg så langt ikke har spesifisert, men som det bør være muligheter for i en default er å ha mulig til å skille på meldingstyper. Altså hvis man lager seg et meldingshierarki (se figur 6-9), så kan man bygge opp alt konstruksjonen med de spesifikke unntakene på toppen og mer generelle nedover, tilsvarende hva man gjør i Java, og eventuelt slutte med en * for å fange opp alt.



Figur 6-9 Eksempel på hierarki av unntaksmeldinger

Dette er så langt utelatt fordi det igjen byr på utfordringer med tanke på gate matching og ikke minst hvordan meldinger skal prioriteres. Med det prioritet så menger jeg at man må være i stant til å hindrer at en default tar meldinger som ikke var tiltenkt den.

6.3. Oppsummering

Jeg har her vist hvordan man systematisk går gjennom sekvensdiagrammer for å avdekke unntak, og strukturere disse funnene. Videre viste jeg hvordan man avgjør hvor unntak skal fanges og håndteres. Jeg har her utelatt kompositt struktren for JavaChatServer, men den vil være tilsvarende hva som var vist i figur 2-3, men med en ekstra etasje for unntakene.

Metodikken har mye til felles med en HazOp analyse, ettersom unntakshåndtering generelt har en del med sikkerhetsanalyse å gjøre. Med det så mener jeg at begge sikter mot å gjøre systemene sikrere og/eller mer robuste.

Metodikken er en intuitiv og enkel og bør aktivt følges når man jobber med å innføre unntakshåndtering i sekvensdiagrammer.

7. Konklusjon

Oppgaven har vært svært omfattende og har bestått av flere deler. Problemstillingen har spent over følgende områder:

- Hva unntakshåndtering er
- Eventuell forskjell mellom unntakshåndtering i modellering og programmering
- Mekanismer for unntakshåndtering og deres presise definisjon
- Metodikk for bruk av mekanismene for unntakshåndtering.

Dette er alle punkter som henger nøye sammen. I første omgang så jeg på hva unntakshåndtering er, og hva eksisterende litteratur sier om dette. Jeg konkluderte med at mekanismene for unntakshåndtering som ble innført i C++ er gjeldene standard i programmeringsspråk, og at unntak handler om uventet oppførsel som må håndteres for å få robuste systemer.

Videre så jeg på forskjellene mellom Java og UML 2 sine sekvensdiagrammer. Jeg sa at Java sin unntakshåndtering er knyttet opp mot hver enkelt tråd sin kall stakk, og ettersom sekvensdiagrammer ikke har noen kall stakk. Derfor er det vanskelig å implementere unntakshåndtering på en tilsvarende måte i sekvensdiagrammer. Konklusjonen her ble likevel at unntakshåndtering generelt handler om det samme, enten det er programmering eller modellering, nemlig det å håndtere uventet oppførsel. De tekniske aspektene ved implementasjonen er det som blir forskjellig.

Deretter presenterte jeg mekanismene for unntakshåndtering. Disse baserte seg på en planløsning og sammen med bruk av U2TP sine defaults. Planløsningen løste utfordringen med gate problematikken, mens defaults blant annet gav en robust måte å ta i mot unntak på. Til sammen gav dette også en mulighet for løs kobling mellom sending og mottak av unntak, til tross for mangel på en kall stakk.

Jeg tok deretter for meg den presise semantikken for mekanismene. I den forbindelse så jeg også på ulike problemstillinger knyttet til tracene. Spesielt var det store utfordringer knyttet til den økte graden av parallellitet som unntakshåndteringen i gitte situasjoner kan medføre.

Til slutt presenterte jeg en enkel og intuitiv metodikk for unntakshåndtering. Denne metodikken tok utgangspunkt i HazOp analyse, som er godt kjent fra sikkerhetsanalyse. Metodikken gav en strukturert måte å arbeide seg gjennom sekvensdiagrammer på, for å finne potensielle unntak og hvordan man kunne strukturere disse for til slutt å implementere de i modellen.

For å oppsummere så har jeg gitt svar på de ulike delene av problemstillingen, og vil her spesielt trekke fram forslaget til mekanismer for unntakshåndtering. Mekanismene representerer en mulig løsning for unntakshåndtering i sekvensdiagrammer. Det er fleksible og robuste mekanismer, som framhever normalflyten. Mekanismene har også en tilhørende presis semantikk som underbygger mekanismene.

Det er fremdeles noen løse tråder, spesielt med tanke på å få en bedre oversikt over tracene for unntakshåndtering. Sett i forhold til omfanget og tiden som var til rådighet er jeg godt fornøyd med det endelige resultatet.

Litteratur

Deitel, H. and P. Deitel (2002). Java How To Program, Fourth Edition. New Jersey, Prentice Hall.

Ghezzi, C. and M. Jazayeri (1997). Programming language concepts, John Wiley & Sons, Inc.

Goodenough, J. B. (1975). "Exception handling: issues and a proposed notation." Communications of the ACM 18(12): 683 - 696.

Gosling, J., B. Joy, et al. (2000). Java(TM) Language Specification (2nd Edition), Addison-Wesley.

Haugen, Ø. and K. Stølen (2003). "STAIRS - Steps To Analyze Interactions with Refinement Semantics." Lecture Notes in Computer Science 2063: 388-402.

Larman, C. (2002). Applying UML and Patterns, Prentice Hall PTR.

Lindholm, T. and F. Yellin (1999). The Java Virtual Machine Specification, Second Edition, Addison-Wesley Professional.

Miller, R. (2004). What's New in UML 2? Model Exceptions. URL: http://bdn.borland.com/article/0,1410,30169,00.html (17.02.2005)

Miller, R. and A. Tripathi (1997). Issues with Exception Handling in Object-Oriented Systems. ECOOP '97 - Object-Oriented Programming: 11th European Conference, Jyväskylä, Finnland, Springer-Verlag Heidelberg.

OMG (2004) UML 2.0 Testing Profile Final Adopted Specification

OMG (2004). Unified Modeling Language: Superstructure.

Redmill, F., M. Chudleigh, et al. (1999). System Safety: HAZOP and Software HAZOP, John Wiley and Sons Ltd.

Rumbaugh, J., I. Jacobson, et al. (2004). The Unified Modeling Language Reference Manual, Second Edition, Addison-Wesley.

Stroustrup, B. (1997). The C++ Programming Language, Third Edition, Addison-Wesley.

Störrle, H., "Semantics of Exceptions in UML 2.0 Activities", Technical Report 0402, *University of Munich*, Germany, 2004.

Winograd, T. (1979). "Beyond Programming Languages." Communications of the ACM 22(7).